Developer Training

Symbian OS Basics

Course Pack

Course Number: 04300

This course pack contains:

- Workbook

NOKIA

BO-EN-04300-20080108

[NOKIA OFFICAL COURSEWARE]

IMPORTANT: READ CAREFULLY BEFORE INSTALLING, DOWNLOADING, OR USING THE NOKIA OFFICIAL COURSEWARE

NOKIA END-USER AGREEMENT

This Nokia End-User Agreement (this "**Agreement**") is between "**You**" the end user (either an individual or an entity), and Nokia Corp. ("**Nokia**"), the licensor. This Agreement authorizes You to use the Nokia Official Courseware specified in Section 1 below, which may be stored on a CD-ROM, sent to You by electronic mail or in written hardcopy, or downloaded from Nokia's Web pages or Servers or from other sources under the terms and conditions set forth below. If You download the Software from a CD-ROM, then the legal notices and any other terms stated therein (collectively, the "**CD-ROM Terms**") shall be incorporated into and made a part of this Agreement by this reference and also govern your downloading and usage of the Nokia Official Courseware. This is an agreement providing certain limited end-user rights and not an agreement for sale. Nokia continues to own the copy of the Nokia Official Courseware You receive and the physical media contained in any package and any other copy that You are authorized to make pursuant to this Agreement.

Read this Agreement carefully before installing, downloading, or using the Nokia Official Courseware. By breaking the seal of the Nokia Official Courseware packet or clicking on the "I Accept" button while installing, downloading, and/or using the Software, You agree to the terms and conditions of this Agreement. If You do not agree to all of the terms and conditions of this Agreement, promptly click the "Decline" or "I Do Not Accept" button, cancel the installation or downloading, or destroy or return the Nokia Official Courseware and accompanying documentation to Nokia. YOU AGREE THAT YOUR USE OF THE NOKIA OFFICIAL COURSEWARE CONSTITUTES YOUR ACKNOWLEDGEMENT THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT, AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS.

1. NOKIA OFFICIAL COURSEWARE. As used in this Agreement, the term "**Nokia Official Courseware**" means, collectively: (i) any software identified in this Agreement, (ii) all contents of the disk(s), CD-ROM(s), electronic mail and its file attachments, or other media with which this Agreement is provided, including the object code form of the software delivered via a CD-ROM, electronic mail, or Web page, (iii) digital images, video presentations, stock photographs, clip art, or other artistic works ("**Stock Files**"), (iv) related explanatory written materials and any other possible documentation related thereto ("**Documentation**"), (v) fonts, (vi) upgrades, modified versions, updates, additions, and copies of the Nokia Official Courseware (collectively "**Updates**"), if any, and (vii) any printed or written materials provided and licensed to You by Nokia under this Agreement.

2. END-USER RIGHTS AND USE. Nokia grants to You a non-exclusive, royalty-free, non-transferable right to use and install the Nokia Official Courseware on the local hard disk(s) or other permanent storage media of one computer and use the Nokia Official Courseware on a single computer or terminal at a time.

3. LIMITATIONS ON END-USER RIGHTS. You may not copy, distribute, or make derivative works of the Nokia Official Courseware, except as follows:

(a) You may make one copy of the Nokia Official Courseware on magnetic media as an archival backup copy, provided Your archival backup copy is not installed or used on any computer. Any other copies You make of the Nokia Official Courseware are in violation of this Agreement.

(b) You may not use, modify, translate, reproduce, or transfer the right to use the Nokia Official Courseware or copy the Nokia Official Courseware except as expressly provided in this Agreement.

(c) You may not distribute, sale, resell, sublicense, rent, lease, or lend the Nokia Official Courseware or any part thereof.

(d) You may not reverse engineer, reverse compile, disassemble, or otherwise attempt to discover the source code of any software included in the Nokia Official Courseware (except to the extent that this restriction is expressly prohibited by law) or create derivative works based on the Nokia Official Courseware.

(e) Unless stated otherwise in the Documentation, You shall not display, modify, reproduce, or distribute any of the Stock Files included with the Nokia Official Courseware. In the event that the Documentation allows You to display the Stock Files, You shall not distribute the Stock Files on a stand-alone basis, i.e., in circumstances in which the Stock Files constitute the primary value of the product being distributed. You should review the "Readme" files associated with the Stock Files that You use to ascertain what rights You have with respect to such mate-

rials. Stock Files may not be used in the production of libelous, defamatory, fraudulent, infringing, lewd, obscene, or pornographic material or in any otherwise illegal manner. You may not register or claim any rights in the Stock Files or derivative works thereof.

(f) You may not use the Nokia Official Courseware to provide training in any class that is not a certified Nokia Course.

(g) You agree that You shall only use the Nokia Official Courseware in a manner that complies with all applicable laws in the jurisdiction in which You use the Nokia Official Courseware, including, but not limited to, applicable restrictions concerning copyright and other intellectual property rights.

4. COPYRIGHT. The Nokia Official Courseware and all rights, without limitation including proprietary rights therein, are owned by Nokia and/or its licensors and affiliates and are protected by international treaty provisions and all other applicable national laws of the country in which it is being used. The structure, organization, and code of the Nokia Official Courseware are the valuable trade secrets and confidential information of Nokia and/or its licensors and affiliates. You must not copy the Nokia Official Courseware, except as set forth in Section 3 above. Any copies that You are permitted to make pursuant to this Agreement must contain the same copyright and other proprietary notices that appear on the Nokia Official Courseware.

5. MULTIPLE ENVIRONMENT SOFTWARE / MULTIPLE LANGUAGE SOFTWARE / DUAL MEDIA SOFTWARE / MULTIPLE COPIES / UPDATES. If the Nokia Official Courseware supports multiple platforms or languages, or if You receive the Nokia Official Courseware on multiple media, or if You otherwise receive multiple copies of the Nokia Official Courseware, the number of computers on which all versions of the Nokia Official Courseware are installed shall be one computer. You may not rent, lease, sublicense, lend, or transfer versions or copies of the Nokia Official Courseware that You do not use. If the Nokia Official Courseware is an Update to a previous version of the Nokia Official Courseware, You must possess valid end-user rights to such a previous version in order to use the Update, and You may use the previous version for ninety (90) days after You receive the Update in order to assist You in the transition to the Update. After such time, You no longer have a right to use the previous version, except for the sole purpose of enabling You to install the Update.

6. THIRD-PARTY LINKS. THE NOKIA OFFICIAL COURSEWARE MAY INCLUDE LINKS TO THIRD PARTY SITES. SUCH LINKED SITES, IF ANY, ARE NOT UNDER THE CONTROL OF NOKIA AND NOKIA IS NOT RESPONSIBLE FOR THEIR CONTENTS, OR ANY OTHER LINK CONTAINED THEREIN. NOKIA PROVIDES THESE LINKS ONLY AS A CONVENIENCE, AND THE INCLUSION OF ANY SUCH LINK DOES NOT IMPLY ENDORSEMENT BY NOKIA OF THE LINKED SITE.

7. COMMENCEMENT & TERMINATION. This Agreement is effective from the first date that You install the Nokia Official Courseware. You may terminate this Agreement at any time by permanently deleting destroying, and returning, at Your own costs, the Nokia Official Courseware, all backup copies, and all related materials provided by Nokia. Your end-user rights automatically and immediately terminate without notice from Nokia if You fail to comply with any provision of this Agreement. In such an event, You must immediately delete, destroy, or return at Your own cost, the Nokia Official Courseware, all back copies, and all related material to Nokia.

8. NO WARRANTY. YOU ACKNOWLEDGE THAT THE NOKIA OFFICIAL COURSEWARE IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, NEITHER NOKIA, ITS LICENSORS OR AFFILIATES, NOR THE COPYRIGHT HOLDERS MAKE AN REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE OR THAT THE SOFTWARE WI NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS, OR OTHER RIGHTS. THE NOKIA OFFICIAL COURSEWARE, ANY DOCUMENTATION AND OTHER INFORMATION, MAY INCLUDE TECHNI- CAL INACCURACIES OR TYPOGRAPHICAL ERRORS. THERE IS NO WARRANTY BY NOKIA OR BY ANY OTHER PARTY THAT THE OPERATION OF ANY SOFTWARE INCLUDED WITH THE NOKIA OFFICIAL COURSEWARE WILL BE UNINTERRUPTED OR ERROR-FREE.

9. NO OTHER OBLIGATIONS. This Agreement creates no obligations on the part of Nokia other than as specifically set forth herein.

10. LIMITATION OF LIABILITY. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL NOKIA, ITS EMPLOYEES OR LICENSORS OR AFFILIATES BE LIABLE FOR ANY LOST PROFITS, REVENUE, SALES, DATA, OR COSTS OF PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, PROPERTY DAMAGE, PERSONAL INJURY, INTERRUPTION OF BUSINESS, LOSS OF BUSINESS INFORMATION, OR FOR ANY SPECIAL, DIRECT, INDIRECT, INCIDENTAL, ECONOMIC, COVER, PUNITIVE, SPECIAL, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND WHETHER ARISING UNDER CONTRACT, TORT, NEGLIGENCE, OR OTHER THEORY OF LIABILITY ARISING OUT OF THE USE OF OR INABILITY TO USE THE SOFTWARE, EVEN IF NOKIA OR ITS LICENSORS OR AFFILIATES ARE ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. BECAUSE SOME COUNTRIES/STATES/JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF LIABILITY, BUT MAY ALLOW LIABILITY TO BE LIMITED, IN SUCH CASES, NOKIA, ITS EMPLOYEES OR LICENSORS OR AFFILIATES' LIABILITY SHALL BE LIMITED TO U.S. $50.

Nothing contained in this Agreement shall prejudice the statutory rights of any party dealing as a consumer. Nothing contained in this Agreement limits Nokia's liability to You in the event of death or per-

sonal injury resulting from Nokia's negligence. Nokia is acting on behalf of its employees and licensors or affiliates for the purpose of disclaiming, excluding, and/or restricting obligations, warranties, and liability as provided in this Section 9, but in no other respects and for no other purpose.

11. TECHNICAL SUPPORT. Nokia has no obligation to furnish You with technical support unless separately agreed to in writing between You and Nokia.

12. EXPORT CONTROL. The Nokia Official Courseware and any related technical data are subject to the U.S. Export Administration Regulations and may be subject to export and import laws, regulations and restrictions in other countries. You agree to comply strictly with all such laws, regulations and restrictions, and acknowledge that You have the responsibility to obtain any licenses or permits that may be required to import, use, export, re-export, or otherwise transfer or disclose of the Nokia Official Courseware or any related technical data. You further agree that except as authorized, You shall not export, re-export or otherwise transfer or disclose of the Nokia Official Courseware, the related technical data or any direct product thereof; (i) into any country embargoed by U.S. export regulations; (ii) to entities controlled by such countries or to nationals of such countries; or (iii) to anyone on the U.S. Treasury Department's list of Specially Designated Nationals and Blocked Persons, the U.S. Commerce Department's Denied Persons List, or the U.S. Commerce Department's Entity List.

13. NOTICES. All notices and return of the Nokia Official Courseware and Documentation should be delivered to:

NOKIA CORPORATION
Forum Nokia Developer Training
Keilalahdentie 4
FIN-02150 Espoo
Finland
Attn: Tero Putkonen

14. GOVERNING LAW, ARBITRATION & GENERAL PROVISIONS.

This Agreement shall be governed by and construed in accordance with the laws of Finland without regard to its conflicts of laws rules.

Any disputes relating to or arising in connection with this Agreement shall be finally settled in arbitration. The arbitrator is to be appointed by the Arbitration Committee of the Central Chamber of Commerce of Finland and the rules of the said Committee are to be followed in the arbitration. The award shall be final and binding and enforceable in any court of competent jurisdiction.

The arbitration shall be held in Helsinki, Finland, in English language.

The parties undertake and agree that all arbitral proceedings conducted with reference to this Agreement shall be kept strictly confidential and all information disclosed in the course of such arbitral proceeding shall be used solely for the purpose of those proceedings.

Notwithstanding the foregoing, nothing in this Agreement shall be deemed to limit the Parties' rights to seek interim injunctive relief or to enforce an arbitration award in any court of law.

If any part of this Agreement is found void and unenforceable, it will not affect the validity of the balance of the Agreement, which shall remain valid and enforceable according to its terms. This Agreement may be modified only in writing by an authorized officer of Nokia.

This is the entire agreement between Nokia and You relating to the Nokia Official Courseware, and it supersedes any prior representations, discussions, undertakings, end-user agreements, communications, or advertising relating to the Nokia Official Courseware.

PLEASE SUBMIT ANY ACCOMPANYING REGISTRATION FORMS TO RECEIVE REGISTRATION BENEFITS WHERE APPLICABLE.

*** End ***

Developer
Training

Symbian OS Basics

Workbook

Course Number: 04300

NOKIA

BO-EN-04300-20080108

Contents

About This Course

Course Description

The goal of this course is to teach students the basics of Symbian programming. The course first shows how to write program code for Symbian OS, and highlights the differences from the ANSI C++ standard. The course also features topics on application structure, the client/server framework and active objects.

Audience

The course is intended for developers who have no prior Symbian C++ development experience but who do have at least 6 months of professional experience developing C++ code for another environment; for example Microsoft Windows or Linux.

Student Prerequisites

Before attending this course, students should typically have:

- Six months experience with programming in C++.

Course Objectives

At the end of the course, students should be able to do the following:

- Explain what Symbian OS is.
- Understand what a UI design platform is and be able to give examples.
- Use the Carbide.c++ IDE to create a new S60 3rd Edition application project.
- Use the Carbide.c++ IDE to import an existing S60 3rd Edition application project from a bld.inf file.
- Use the Symbian OS exception handling mechanism.
- Understand the application framework and what classes are involved.
- Explain what an active object is, and know when to use it and how to implement it.
- Add additional resources to an application.
- Localize the resources of an application.
- Explain what the active scheduler does.
- Explain what a client and a server are and how they communicate within the Symbian OS.

Document Conventions

The following conventions are used in course materials to distinguish elements of the text.

Convention	Use
`Courier10 BT`	Represents code samples or examples of screen text.
|	Separates an either/or choice in syntax statements.
`[ ]`	Encloses optional items in syntax statements. Type only the information within the brackets, not the brackets themselves.
...	Represents an omitted portion of code sample.
`{ }`	Encloses required items in syntax statements. Type only the information within the brackets, not the brackets themselves.
`//`	Distinguishes short comments in C++ and Java programming code.
`//` `//`	Distinguishes long comments in C++ and Java programming code.
`/* ... */`	Distinguishes long comments in C, C++ and Java programming code.
`<!-- ... -->`	Distinguishes comments in HTML, WAP & XHTML programming code.

Module #FN

Forum Nokia

Contents

NOKIA

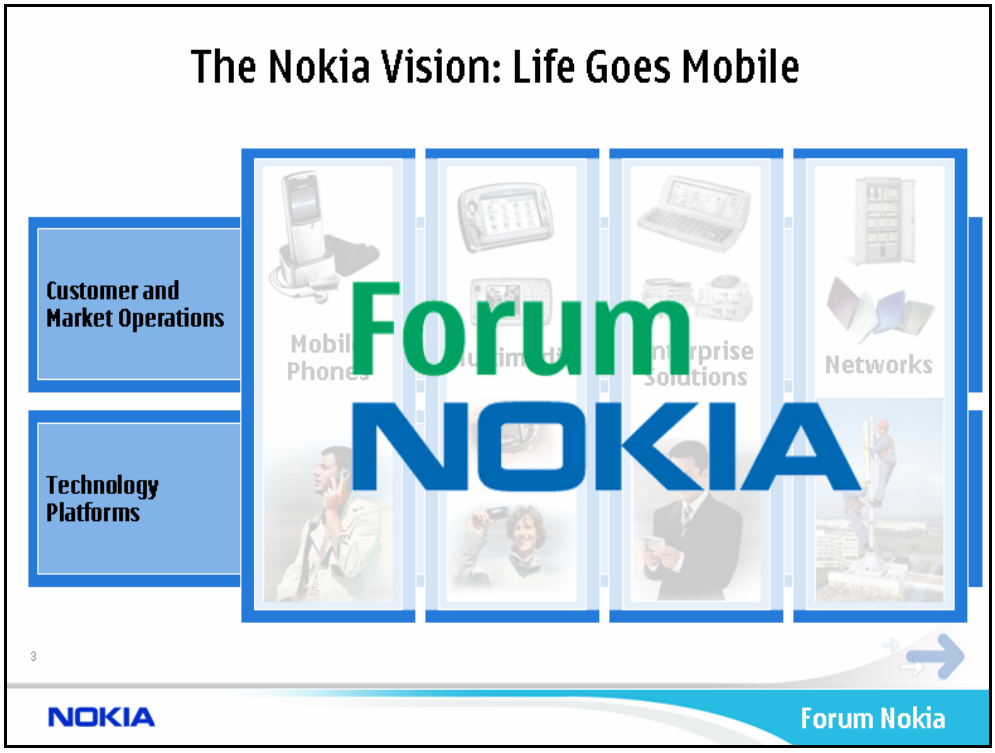
BO-EN-04300-20080108

Forum Nokia

Forum Nokia's Mission

Forum Nokia's mission is to create lucrative business opportunities for mobile developers globally. We connect developers to tools, technical information, support, and distribution channels they can use to build and market applications around the globe. From offices in the U.S., Europe, Japan, China, and Singapore, Forum Nokia provides technical and business development support to developers and operators to assist them in achieving their goal of successfully launching applications and services to consumers and enterprises.

The Nokia Vision: Life Goes Mobile



This slide shows you where we are in the Nokia organization and how we support Nokia's vision '*Life Goes Mobile*'. Nokia sees mobility expanding into new areas such as imaging, games, entertainment, media and enterprises, which gives a lot of opportunities for application developers.

Nokia is organized into four business groups:

- **Mobile Phones** makes user-friendly mobile devices with many features for different segments of the global market.
- **Multimedia** brings connected mobile multimedia to consumers in the form of advanced mobile devices and solutions.
- **Enterprise Solutions** is dedicated to helping businesses improve their performance by extending their use of mobility from mobile devices for voice and basic data to secure mobile access and use of their content and applications.
- **Networks** is a leading provider of network infrastructure, communications and networks service platforms and services to operators and service providers.

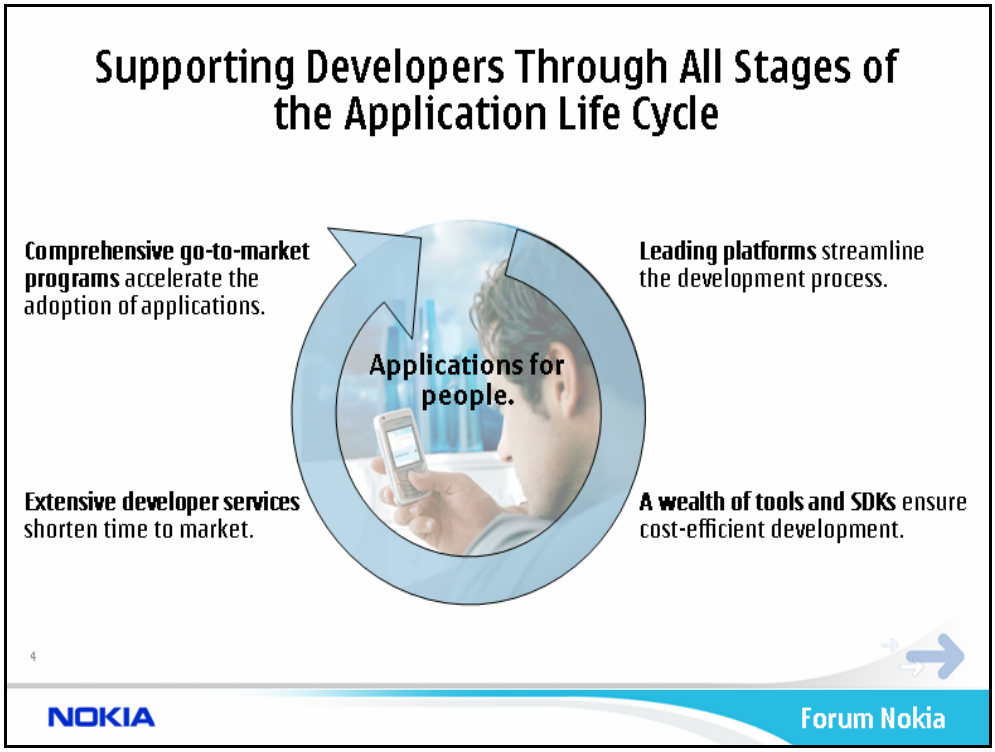
The business groups are supported by **two horizontal organizations**:

- **Customer and Market Operations**, which is responsible for the sales, manufacturing, logistics, and sourcing for Nokia's mobile device businesses.
- **Technology Platforms**, which delivers leading technologies and platforms to Nokia's business groups and external customers.

For developers and content providers, **Forum Nokia provides the front door to working with Nokia at every level.** Forum Nokia cuts across all Nokia Business Groups, supporting their business. It is extremely important for Nokia to work with a rich base of

developers, who can help us ensure that consumers have a wide range of applications and content to enhance and personalize their devices, and businesses can improve their performance through extended mobility. For operators, the consumption of mobile applications and services bring more ARPU (average revenue per user).

Supporting Developers Through All Stages of the Application Life Cycle



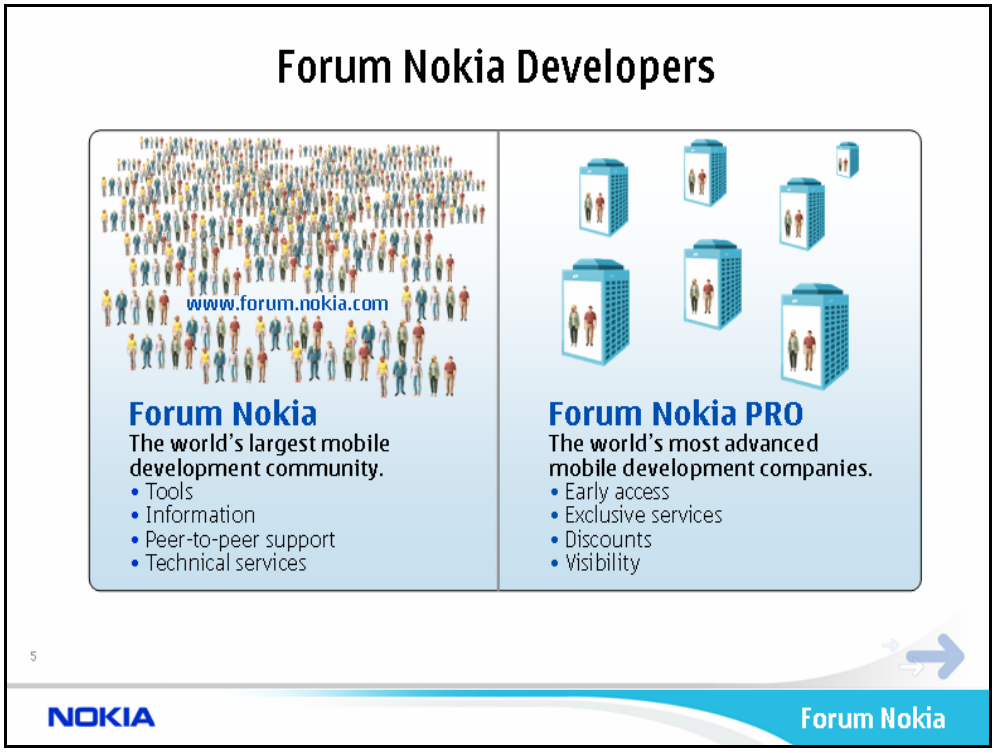
Forum Nokia's mission is to create lucrative business opportunities for mobile developers globally. We do this by supporting developers through all stages of the application life cycle: from developing to selling. We connect more than 2 million people registered at www.forum.nokia.com with tools, technical information, and support. For the selected few, the 400+ Forum Nokia PRO members, we offer exclusive and preferential services.

The **platform approach** and extensive resources covering tools, SDKs, specifications, documentation, technical support etc. help build applications to a global audience in less time, with less effort and at a lower cost.

The Preminet Client Solution and other go-to-market programs help developers take applications and content to customers as well as to businesses.

In order to ensure that our mission is viable, our guiding principle is 'Applications for People'. This idea drives everything we do as an organisation, simply because end users – be them consumers or business users – will have the final word on whether applications and services will fly on the market or not.

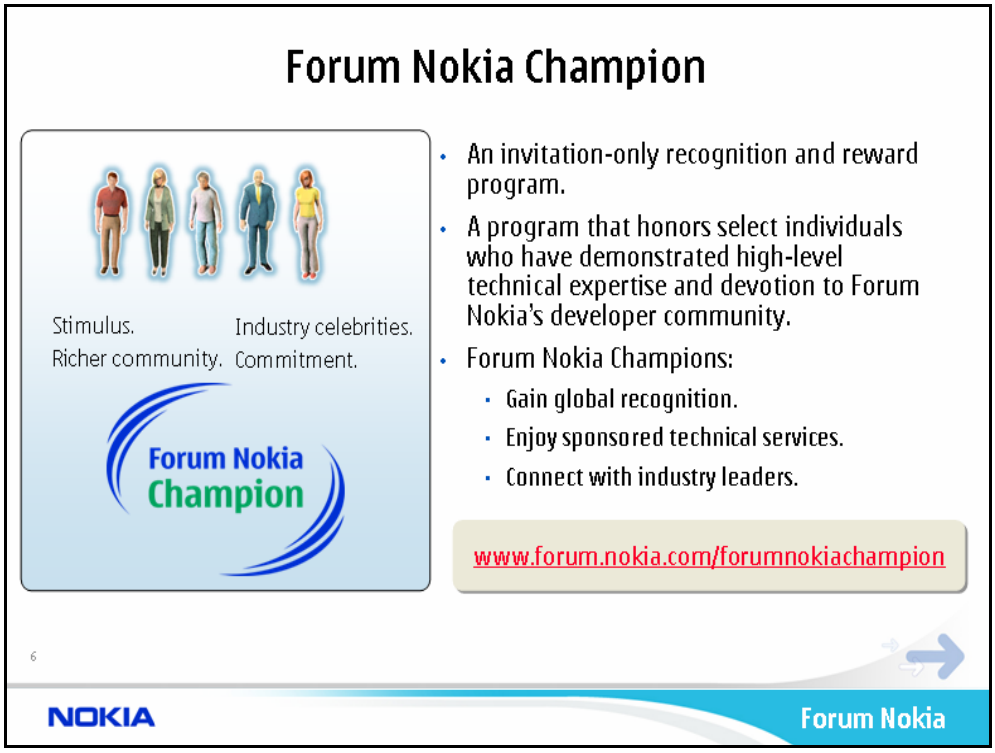
Forum Nokia Developers



Forum Nokia is Nokia's global developer program connecting developers with tools, technical information, documentation, and technical services to help them build and market applications around the globe. Currently, there are **more than 2 million registrants at forum.nokia.com**, which makes Forum Nokia the world's largest dedicated mobile development community. Forum Nokia registrants represent more than 230 countries, territories and colonies around the globe, with the EMEA (Europe, Middle East, and Africa) region accounting for the largest number (about 60%), followed by Asia (about 30%) and the Americas (about 10%).

In February 2004, we kicked off **Forum Nokia PRO**, our premium developer program. Today, we have **more than 400 advanced mobile-developer companies as members**. The member companies get exclusive access to platform roadmaps, early access to devices and SDKs, preferential access to technical training and support services, as well as business development, co-marketing, and media relations support. Each PRO member company has a named contact person in Forum Nokia.

Forum Nokia Champion



Nokia launched on February 9, 2006 Forum Nokia Champion initiative. Forum Nokia Champion is an invitation-only recognition and reward program for top mobile developers worldwide. Every six months, Forum Nokia invites outstanding members of the forum.nokia.com community to accept a one-year membership in this reward program. Honorees are selected on the basis of their technical expertise and their activities and contributions to forum.nokia.com online and offline services and communities. Examples of activities and achievements that may be considered and used as a basis for selection are participation in discussion boards at forum.nokia.com, exposure in public media streams for Nokia platform expertise, developer feedback activity, sharing expertise at educational activities or academic publications, hosting Nokia platform-related Web sites, gaining commercial success and/or getting patents related to Nokia platforms, etc.

Each Forum Nokia Champion earns elevated professional profile and status, sponsored access to Nokia technical services, and exclusive networking opportunities. Enhanced recognition is the key benefit of being a member of Forum Nokia Champion. Forum Nokia Champions have the right to use the Forum Nokia Champion title and qualifier image during their membership year. They get a welcome gift, are showcased as Forum Nokia Champions online, enjoy sponsored technical services, can better connect with industry leaders via various events, and gain exclusive access to the Forum Nokia Champion extranet.

The key difference with Forum Nokia PRO program is that Forum Nokia Champion is targeted only at individual registrants at forum.nokia.com. It is not for companies. For example, a person selected to be a Forum Nokia Champion *may* be an employee of a Forum Nokia PRO company, but he is not selected for his achievements in his work role but based on his personal activities, achievements, and technical expertise.

All the current Forum Nokia Champions who have given their permission are listed at www.forum.nokia.com/forumnokiachampion. We're happy to say that we have received overwhelmingly positive feedback from our developers on this initiative.

The Platform Approach

To help developers build and deliver mobile applications to consumers around the globe in less time, with less effort and at a lower cost, we have adopted a **platform approach**. This means that each platform - Series 40, S60, and Series 80 - incorporates a standard set of technologies, such as Java technology, XHTML browsing, and Multimedia Messaging, on a series of Nokia devices.

With the help of extensive documentation, development tools, and technical services, developers can make their Java and Symbian applications, Web content, and MMS content available across this entire family with minimal optimization effort. Through the platform strategy, Nokia maximizes developers' volume opportunities and minimizes the need to port applications and services to individual handsets. The huge market potential that the Series 40 Platform, the S60 platform, and the Series 80 Platform represent is evident in the light of these figures (from November 2005):

- By the end of 2005, Nokia had cumulatively shipped approximately 350 million devices globally based on Nokia platforms for application development.

Series 40 is the platform for mass-market mobile devices:

- The platform offers seamless application interworking and uniform user experience,
- It's a highly configurable platform: wide range of different product configurations.
- It's inherently secure.
- Now with Flash Lite shipping on some Series 40 devices, there are additional opportunities for developers creating rich mobile applications.

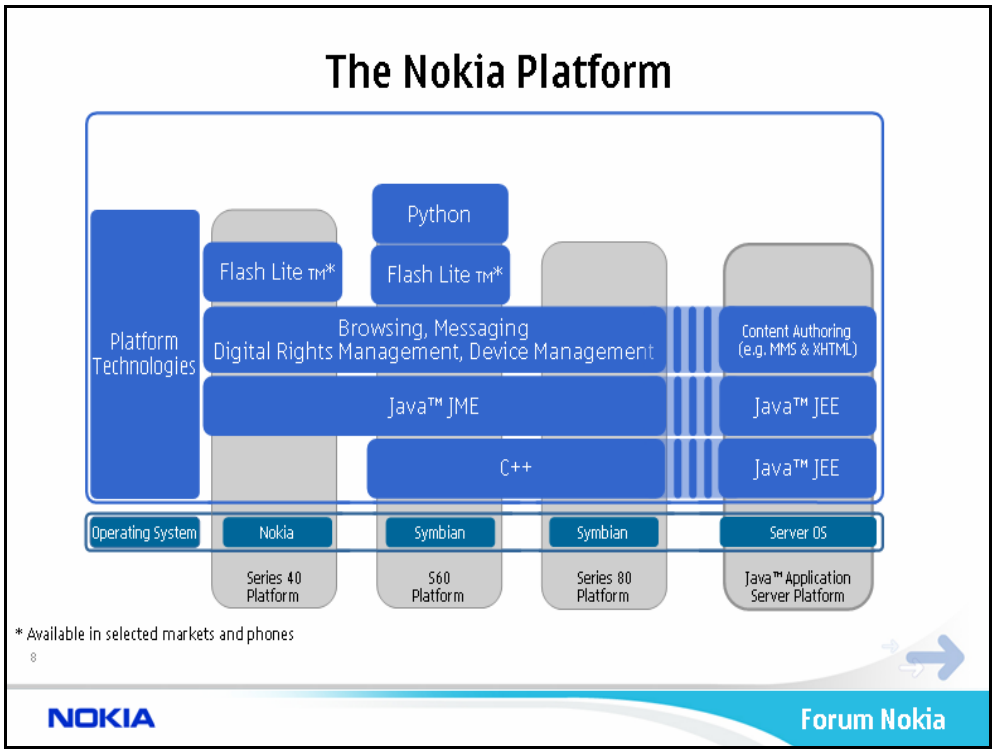
S60, our SmartPhone platform is really heating up.

- Nokia alone will have cumulatively shipped more than 50 million S60 enabled devices by the end of February 2006.
- Smartphone market is expected to continue its fast growth, reaching close to 100 million units during 2006. According to Nokia estimation, bulk of this volume will be based on S60 3rd Edition.
- 30+ S60 devices have been launched by February 2006.
- The S60 Platform is licensed by some of the foremost mobile phone manufacturers.
- Addition of Flash Lite to many S60 devices opens up the mobile development market to Flash developers creating rich mobile applications.

Series 80 is our enterprise platform for Java and C++ applications, as well as rich content for Symbian OS enterprise devices. It provides:

- Advanced connectivity options, device management and improved performance.
- End-to-end security and backend connectivity.

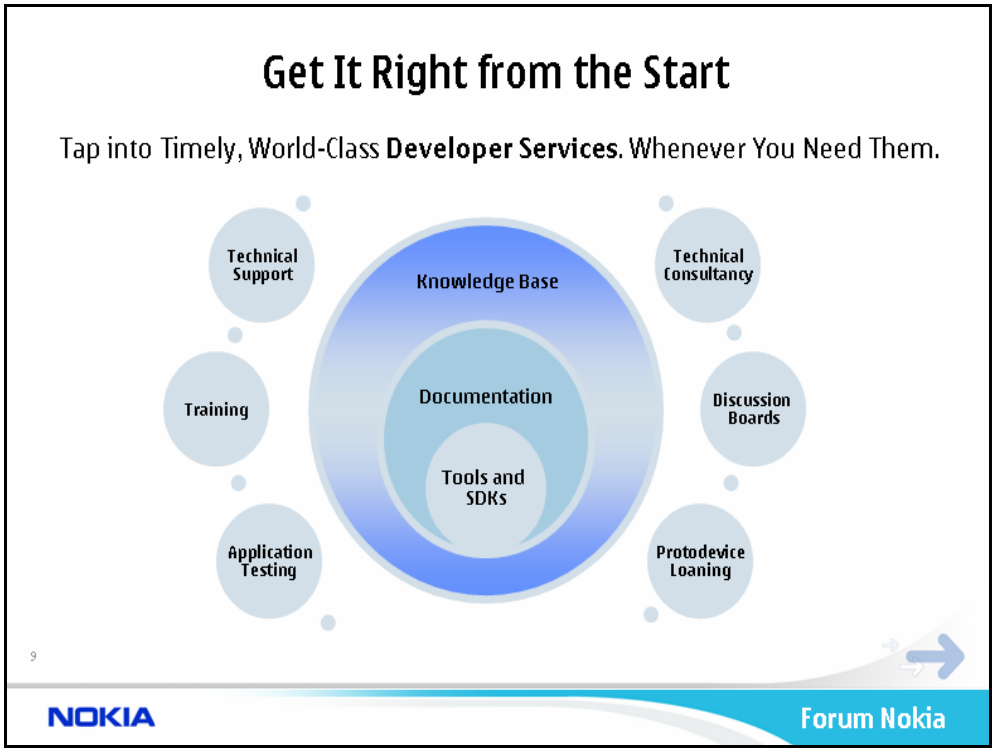
The Nokia Platform



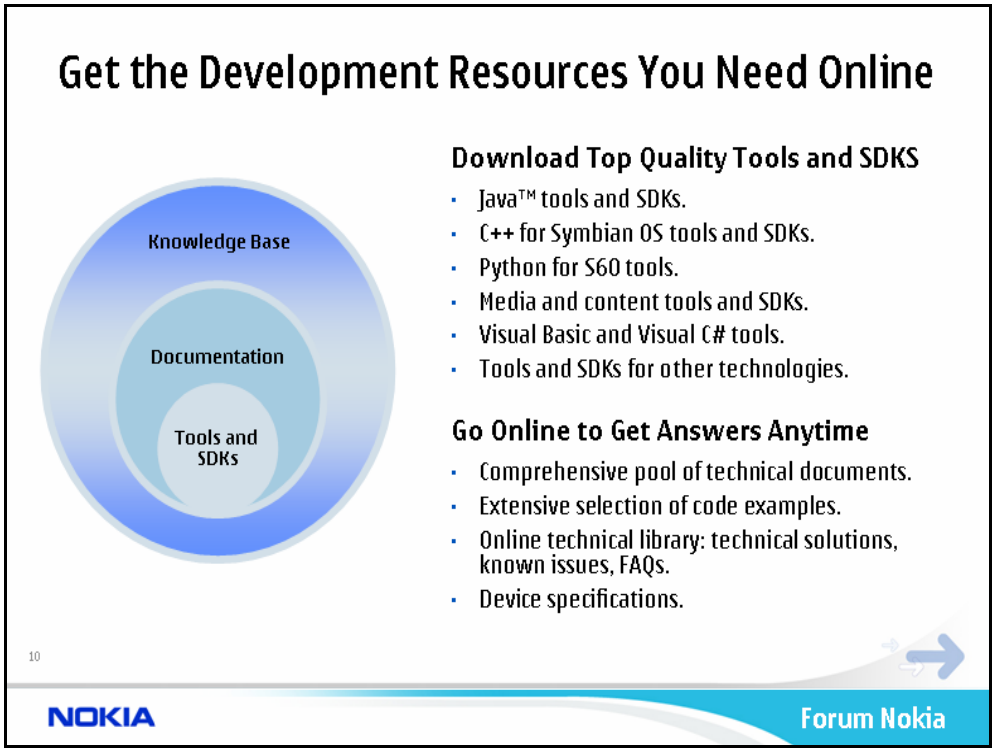
Here, at a glance, is the Nokia platform.

If you have seen this architecture diagram in the past you might notice that it has been updated to include Flash Lite, which is exciting for all members of the Nokia community – developers, operators and end user since it brings a compelling graphical user experience, fast and inexpensive development costs, a vital developer community and a huge market opportunity.

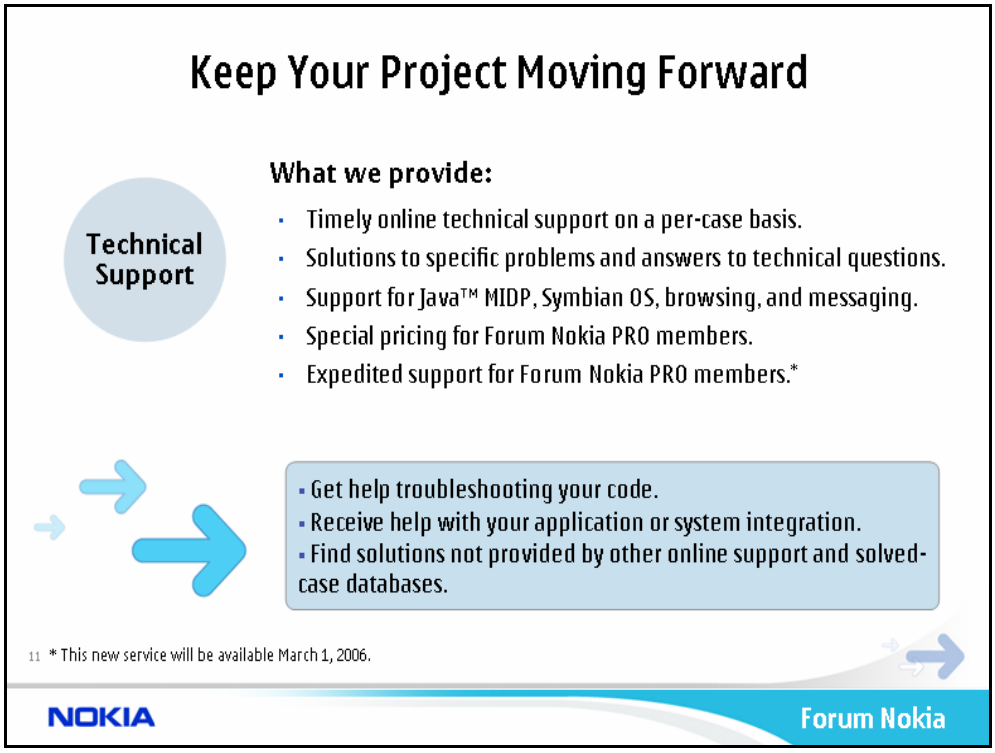
Get It Right from the Start



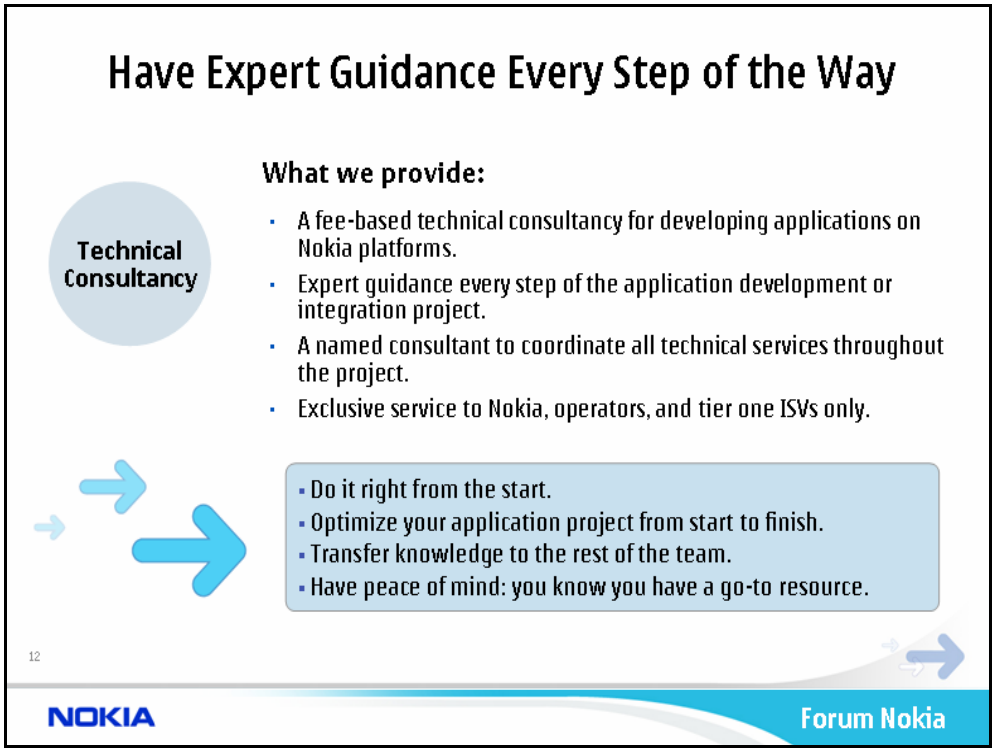
Get the Development Resources You Need Online



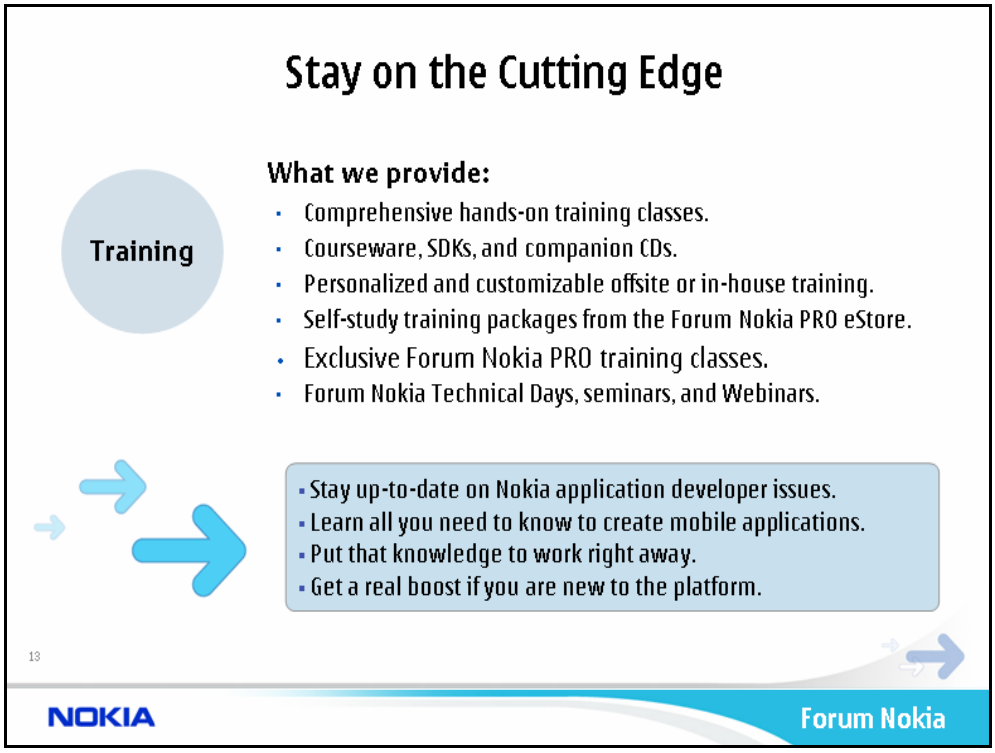
Keep Your Project Moving Forward



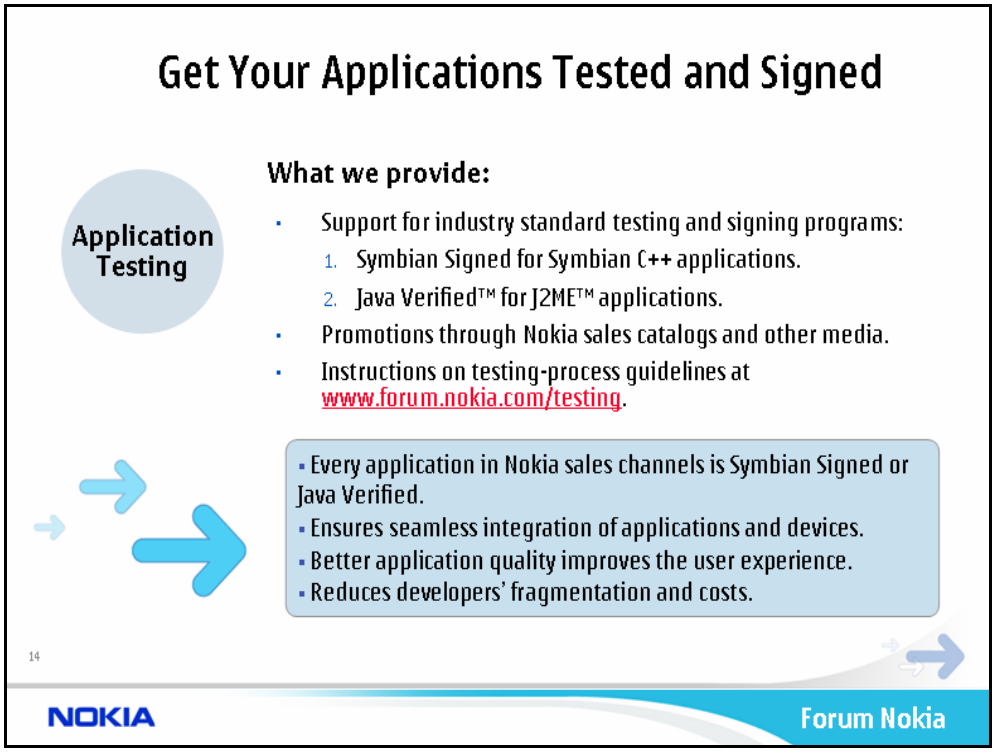
Have Expert Guidance Every Step of the Way



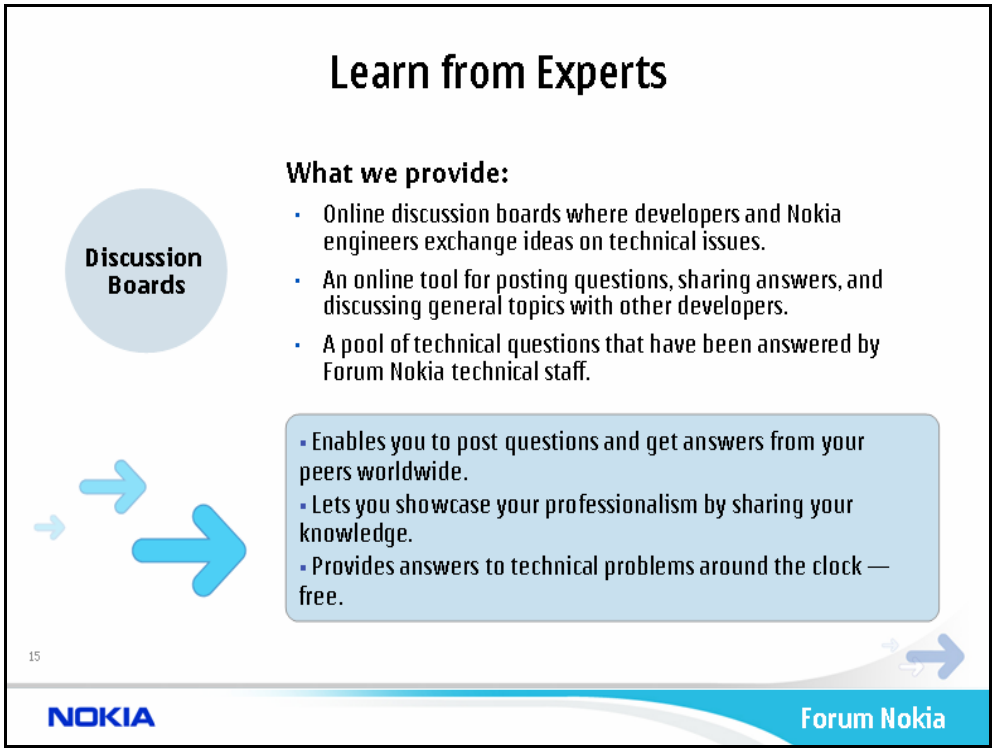
Stay on the Cutting Edge



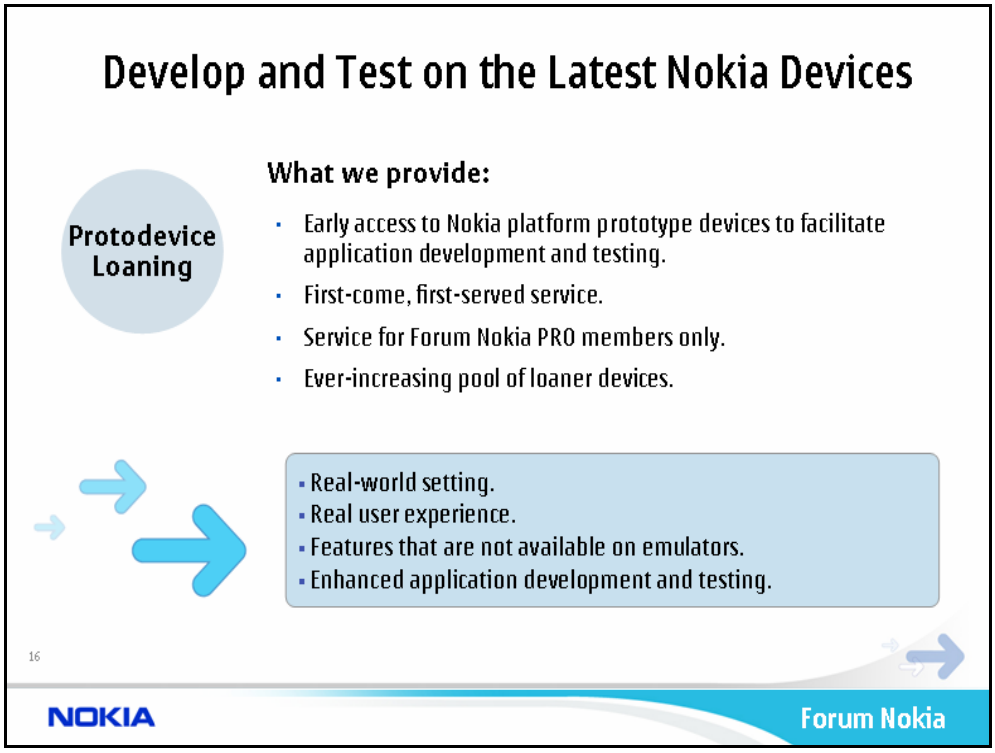
Get Your Applications Tested and Signed



Learn from Experts



Develop and Test on the Latest Nokia Devices



2500+ prototype devices loaned to Forum Nokia PRO members during 2005 *(this is amount of devices that has been shipped to Forum Nokia PRO members during 2005, same device can be loaned multiple times in a year).*

Developer
Training

Module #04301

Symbian OS Background

Contents

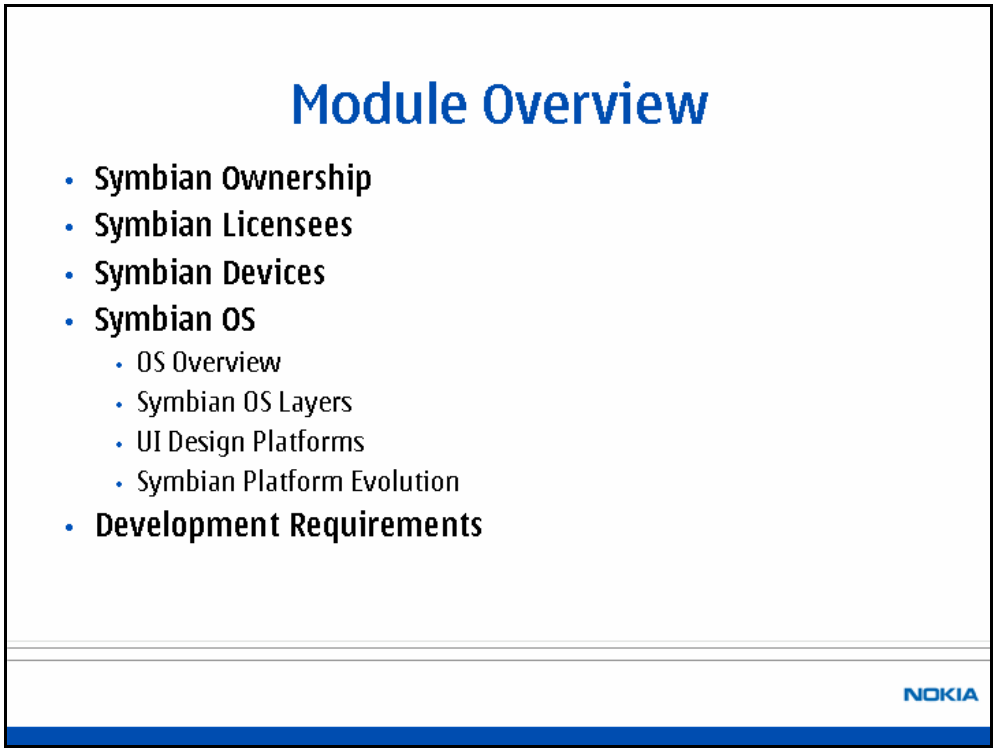
NOKIA

BO-EN-04300-20080108

Symbian OS Background

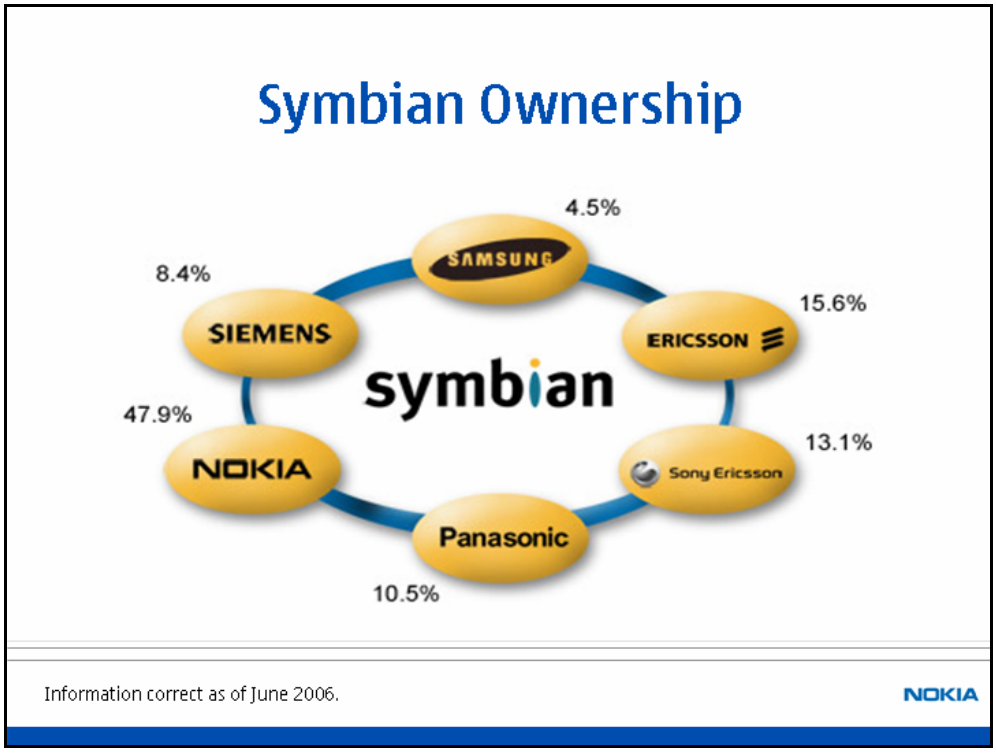
This lesson covers Symbian introduction and history. It provides background information as to who Symbian is, the advantages of the Symbian OS, and the devices that use it.

Module Overview



The first part of the lesson introduces Symbian and the Symbian OS. It covers the companies that make up Symbian and the licensees of the Symbian OS. A summary of devices that use Symbian OS is included before giving an overview of the OS and the advantages of using it.

Symbian Ownership



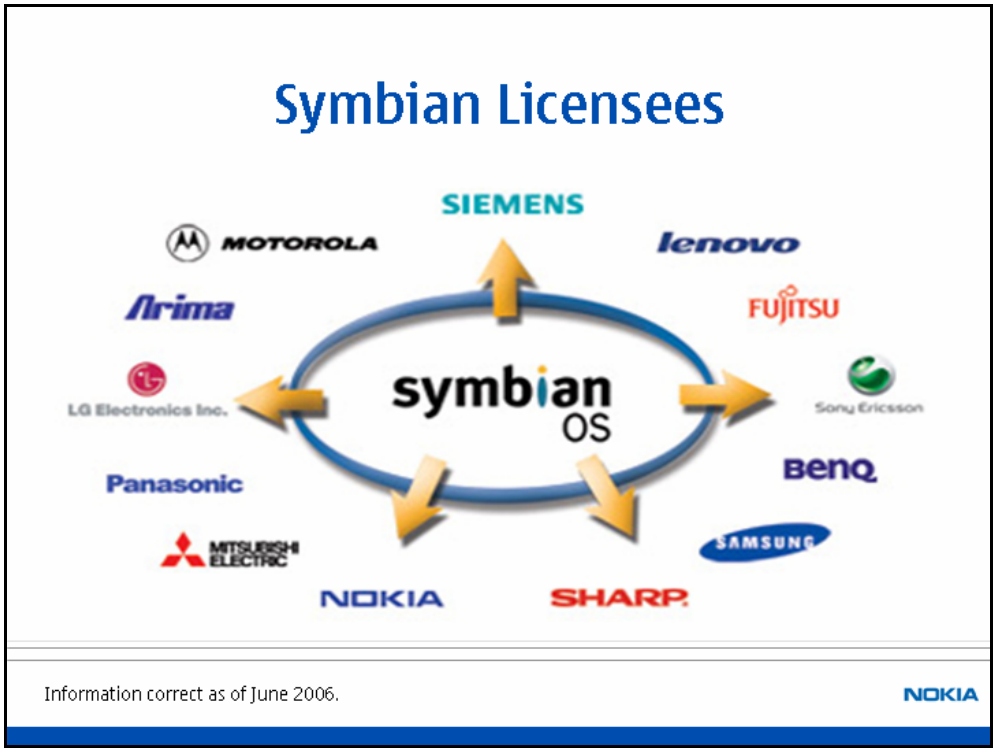
Symbian is a software licensing company; its core business is to supply the advanced, open operating system, Symbian OS, for data-enabled mobile phones. Symbian was established as a private independent company in June 1998. It has its headquarters in the UK, with offices in Japan, Sweden, UK, USA and India. Symbian now has more than 1200 staff.

The companies that own Symbian are shown in the above slide along with the respective percentage of shares each company possesses in Symbian. As you can see in the slide, Nokia own the predominant stake in the company with 47.9% of the shares.

For a brief history of Symbian, visit the following link:

http://www.symbian.com/about/overview/history/history.html

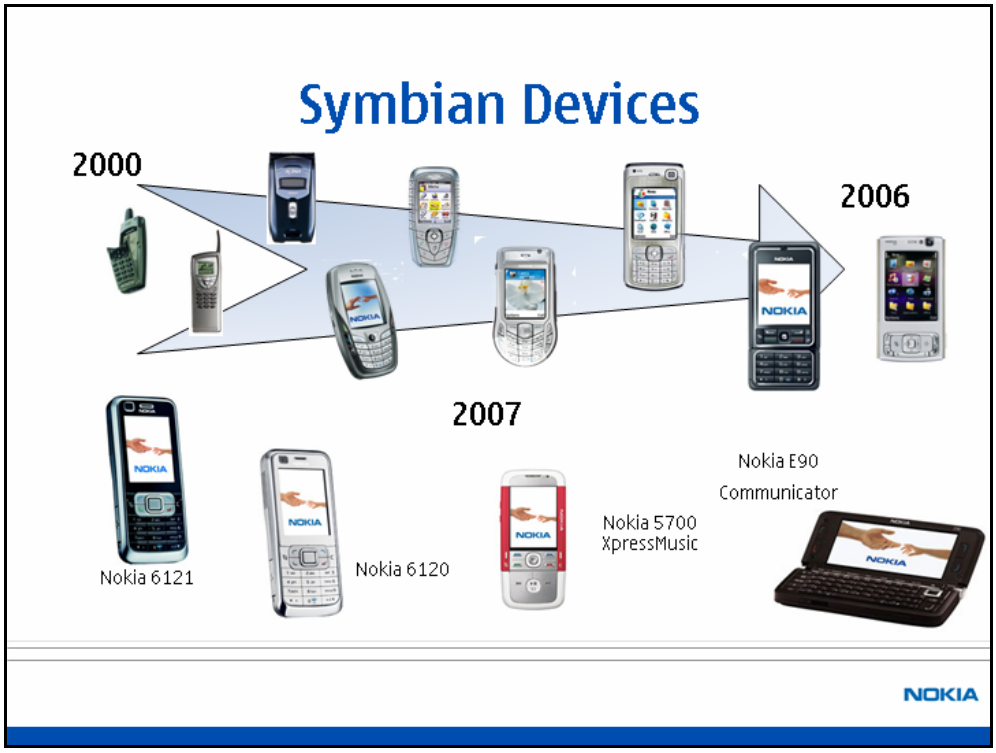
Symbian Licensees



Symbian does not manufacture devices itself. Instead, it licenses its OS to device manufacturers, who then use it in their products. The current Symbian licensees are shown in the above slide.

Note that with the exception of Ericsson, the companies that own Symbian are also Symbian licensees.

Symbian Devices

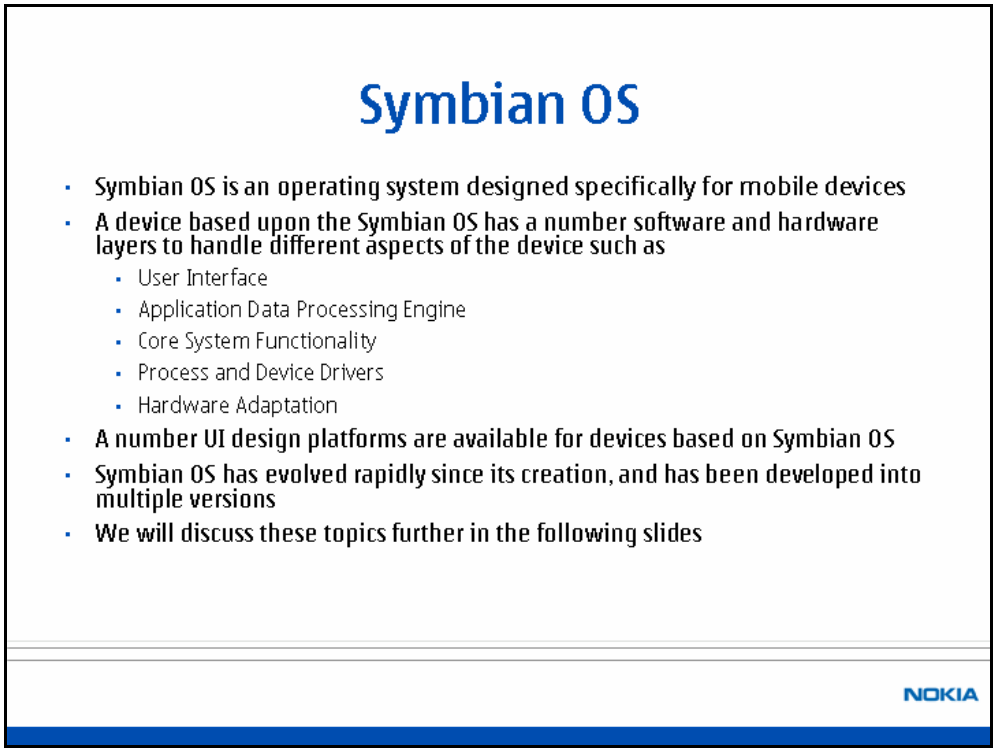


The above slide shows a selection of the Symbian OS devices that have been produced over previous years (from a cross section of manufacturers). The first Symbian OS phone, the Ericsson R380, went on sale to the public in 2000. Since this time a multitude of different types of Symbian OS devices have been produced; with the complexity of the devices increasing year on year.

For a complete list of the different types of Symbian OS devices that have been produced (and information about each type), visit the following link:

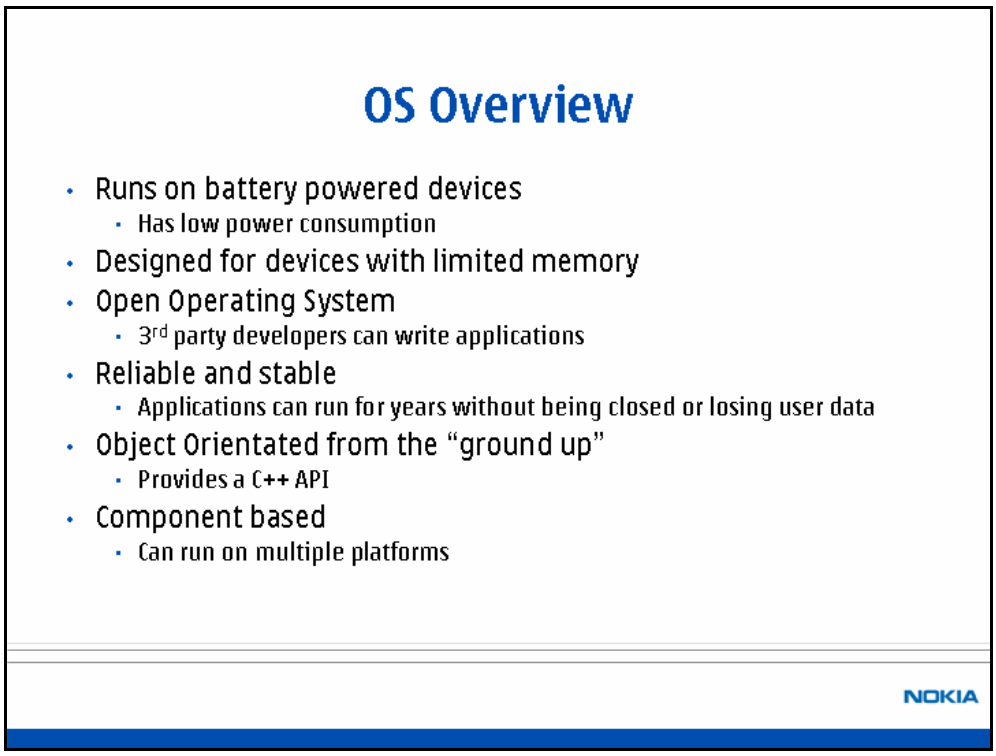
http://www.symbian.com/phones/

Symbian OS



In the next group of slides we will present a brief overview of the key characteristics of the Symbian OS, as well as how it has progressed since it emerged.

OS Overview



The Symbian OS was designed specifically for mobile devices and, as such, has a small memory footprint and low power consumption. This is very important since users do not want to recharge their phone every day.

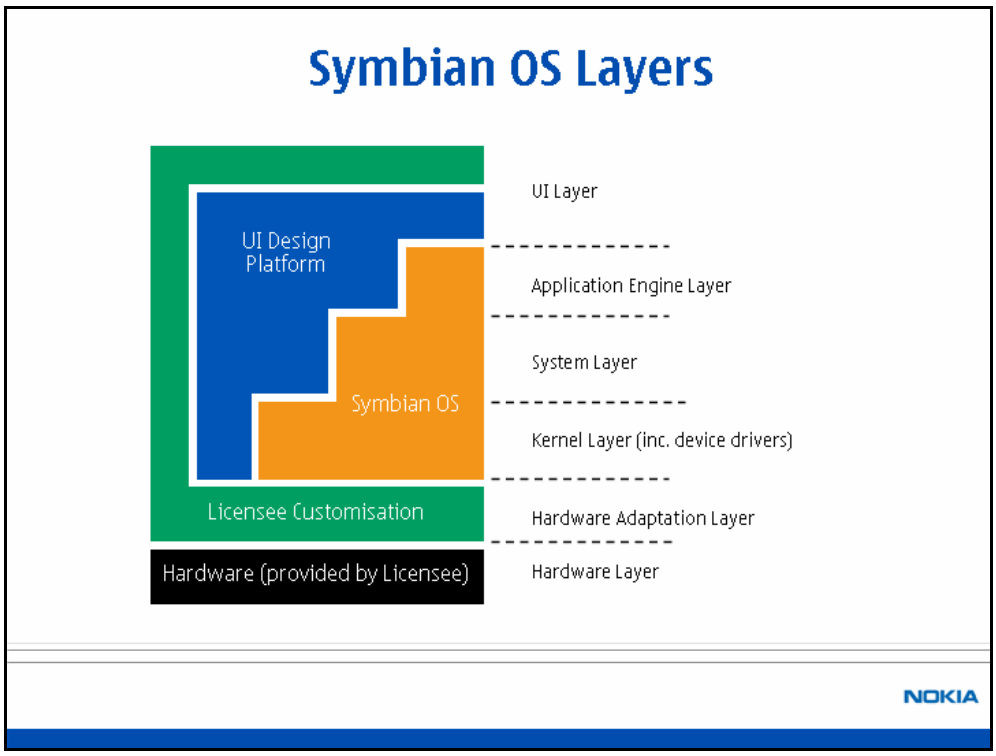
It is an open operating system, enabling third-party developers to write and install applications independently from device manufacturers.

An extensive C++ API is provided which allows access to services such as telephony and messaging, in addition to basic operating system functionality.

Some devices that run the Symbian OS may not be switched off for years; therefore, the operating system was designed so that applications could run for years without losing user data. The operating system can run on more than one hardware platform, so it can be used on a variety of different device types including those with touch screens and those with pens or keyboards.

Symbian OS is the current name of the operating system, but when it was initially released it was known as "EPOC". The name EPOC was used for some time and will still be found in class/file names and in older documentation.

Symbian OS Layers



As can be seen on the slide, the software on a typical Symbian OS device is split into a number of layers. These layers are explained below:

- UI Layer – this contains software specific to the UI; including all the applications.
- Application Engine Layer – provides access to data that is required by applications.
- System layer - provides all the core functionality of the system. This is usually in the form of system servers.
- Kernel Layer – software that provides kernel services; e.g process and thread creation.
- Hardware Adaptation Layer – software required so the upper software layers will run on the chosen hardware. This layer is dependent on the hardware. All upper layers are independent of the hardware.
- Hardware – The actual physical hardware on which the software will run. This is either supplied by the licensee themselves or licensed from an Original Design Manufacturer (ODM).

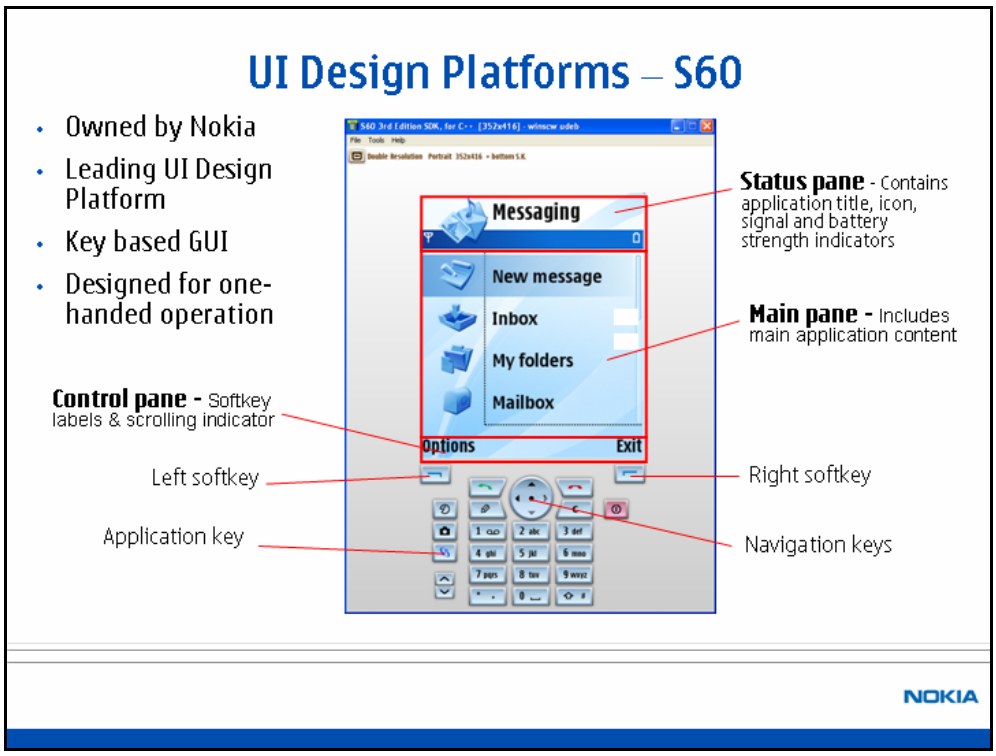
To manufacture a complete phone based on Symbian OS, not only must you have the Symbian OS software but you must have additional UI design platform and customization software. These are described below:

- The UI design platform is software that implements a particular UI design. There are a variety of different platforms available.
- Customization software - distinguishes a device from others in the marketplace; e.g. adding certain functionality not implemented by the Symbian OS or the UI Platform.

UI Design Platforms

This following slides discusses some of the most widely used UI design platforms available for the Symbian OS.

UI Design Platforms – S60



The S60 Platform is the leading UI design platform for Symbian OS and is owned by Nokia and licensed to other device manufacturers including LG Electronics, Lenovo, Nokia, Panasonic, Samsung, Sendo and Siemens.

The S60 UI is designed for one hand operation, is key based and is the most extensively researched and thoroughly developed GUI ever created by Nokia. Its inclusion in the S60 Platform ensures UI consistency across all S60 platform based phones from all device manufacturers.

Later versions of S60 include Scalable UI support for the following screen resolutions (in pixels): 176 x 208 (classic), 240 x 320 (QVGA), and 352 x 416 (double). The Scalable UI architecture also supports each of these screen resolutions in either portrait or landscape view and adopts the scalable vector graphics (SVG) format for icons and themes. S60 also support flexible customisation possibilities for licensees and operators.

As of June 2006, the latest version of S60 is 3rd Edition which is based on Symbian OS v9.1.

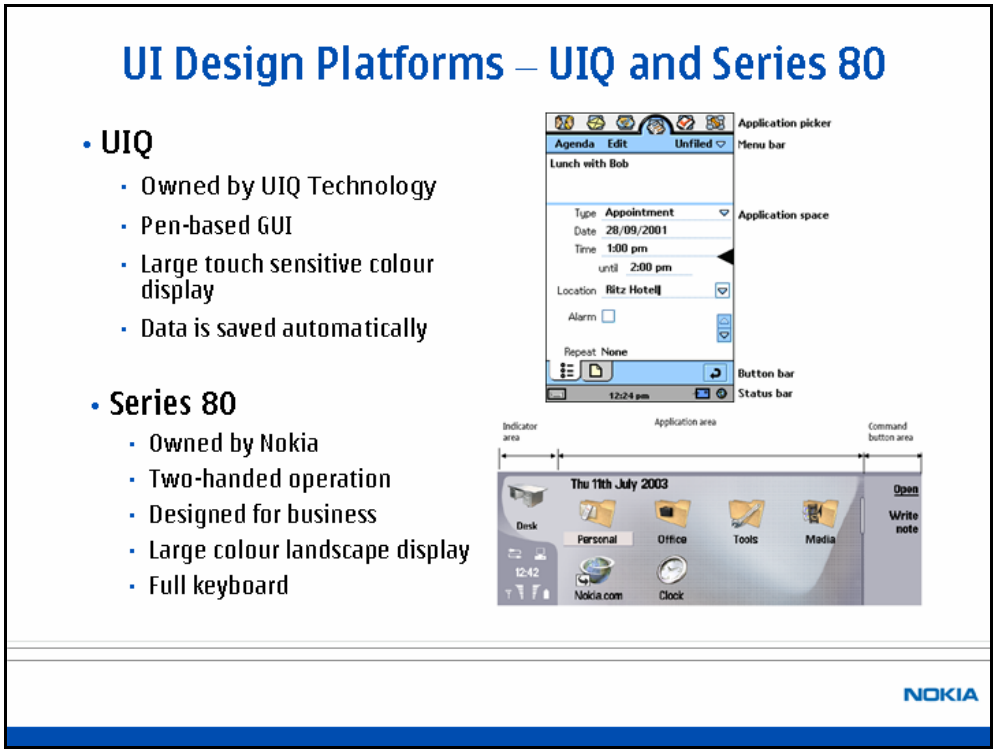
The labs included in this course are associated with components that are compatible with the S60 UI design platform. For this reason a little more detail is given below about the S60 screen layout and S60 specific keys. These details are also indicated in the slide.
The S60 screen is divided into the following areas:

- Status Pane – displays status information of the current application and state, as well as general information about the device status (signal strength, battery charging); consists the following sub-panes: title pane, context pane, navi pane, signal pane and battery pane.
- Main Pane – the principal area of the screen where an application can display its data.
- Control Pane – displays the labels associated with the two softkeys.

Any S60 device must contain a number of S60 specific keys including:

- Left and right softkeys – These perform actions associated with the label displayed in the control pane. For example the left softkey usually opens an options menu and the right softkey usually takes the user back to the last screen or exits an application.
- Applications key – This key toggles between the applications menu (i.e. a screen containing a list of applications) and the main screen.
- Navigation keys – These keys allow the user to navigate up, down, left and right and to select an item. They are used for example in selecting an item from a grid or list.

UI Design Platforms – UIQ and Series 80



UIQ

The UIQ UI design platform is owned by UIQ Technology which is a fully owned subsidiary of Symbian. It is licensed by UIQ Technology to device manufacturers including: Arima, BenQ, Motorola and Sony Ericsson.

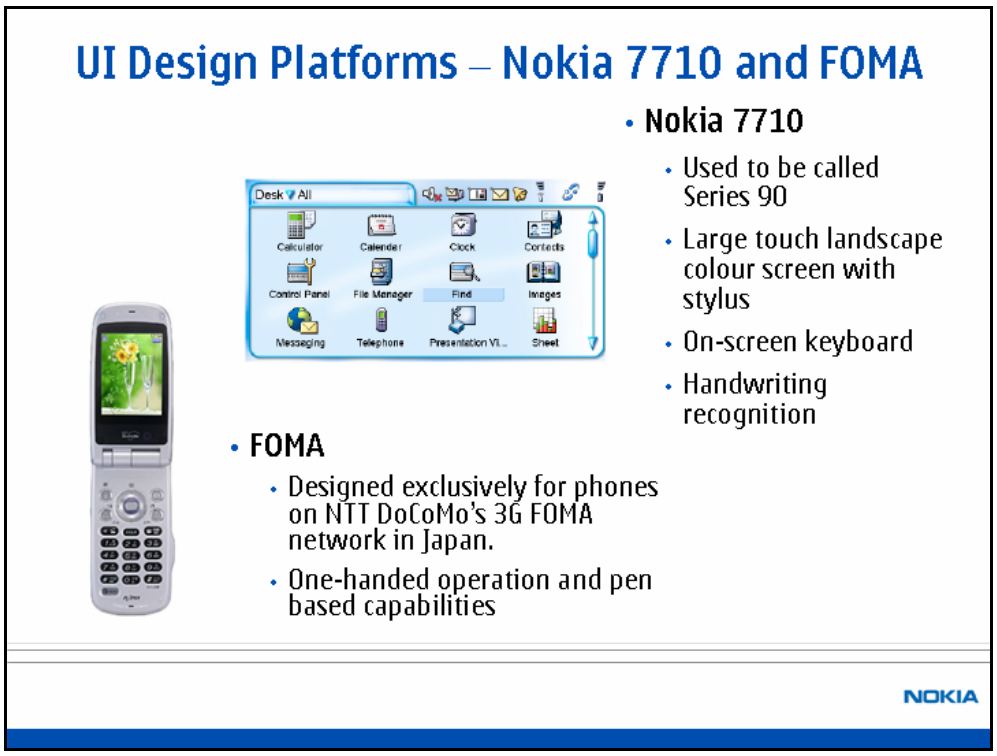
UIQ is a customizable pen-based graphical user interface for media-rich Symbian OS mobile phones. UIQ is designed to make effective use of the phone's large, touch-sensitive colour display and enable richer data services. The focus of the platform is on usability; all changes are automatically saved and an application stays open with its list view displayed for quick navigation, making it simple for users to orient themselves.

Series 80

The Series 80 platform is designed for creating mobile devices for business use. It is owned and used exclusively by Nokia.

The Series 80 platform is designed for two-handed use and makes use of a full QWERTY keyboard. In addition it has a large colour display. Devices that use the Series 80 UI design are the Nokia 9210, Nokia 9500 and Nokia 9300.

UI Design Platforms – Nokia 7710 and FOMA



Nokia 7710

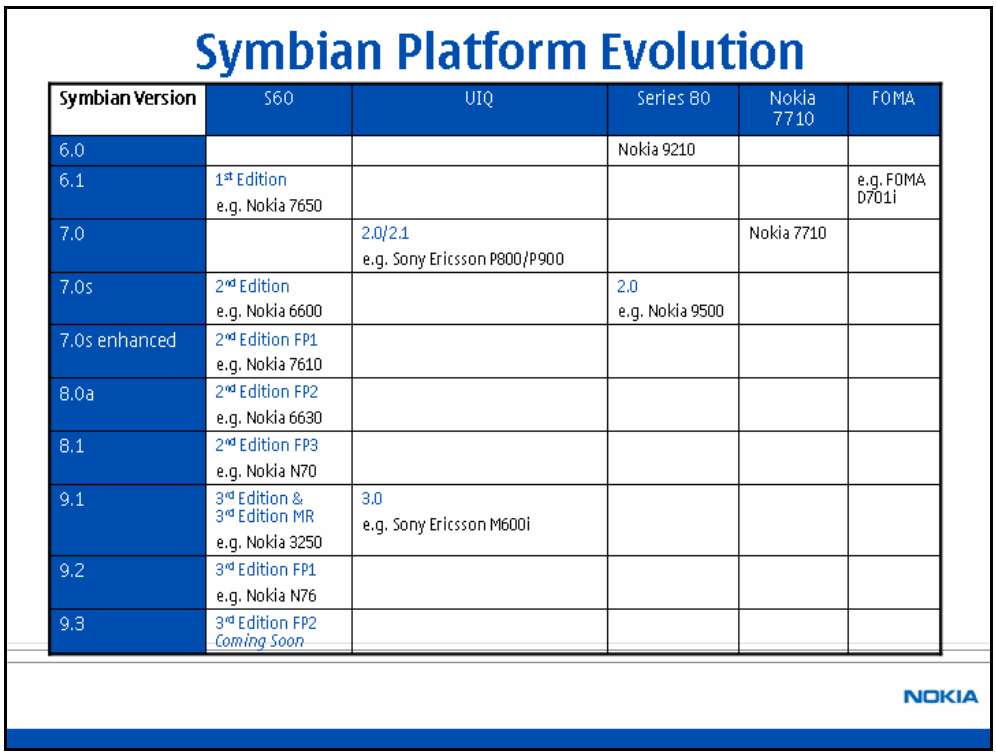
This used to be called Series 90. The Nokia 7710 was the only device ever produced using this UI design before Nokia decided that the design was no longer required. The name of the device is used now instead.

It has a large landscape colour screen that is touch sensitive for use with a stylus. It features on screen keyboard and handwriting recognition.

FOMA

The FOMA SW platform is designed exclusively for phones on NTT DoCoMo's 3G FOMA network in Japan. Symbian OS licensees Fujitsu and Mitsubishi have both implemented the FOMA SW platform in their phones, enabling the fast and seamless delivery of advanced services to NTT DoCoMo customers. FOMA SW platform offers one-handed use as well as pen-based capabilities, featured in the FOMA F900iT.

Symbian Platform Evolution



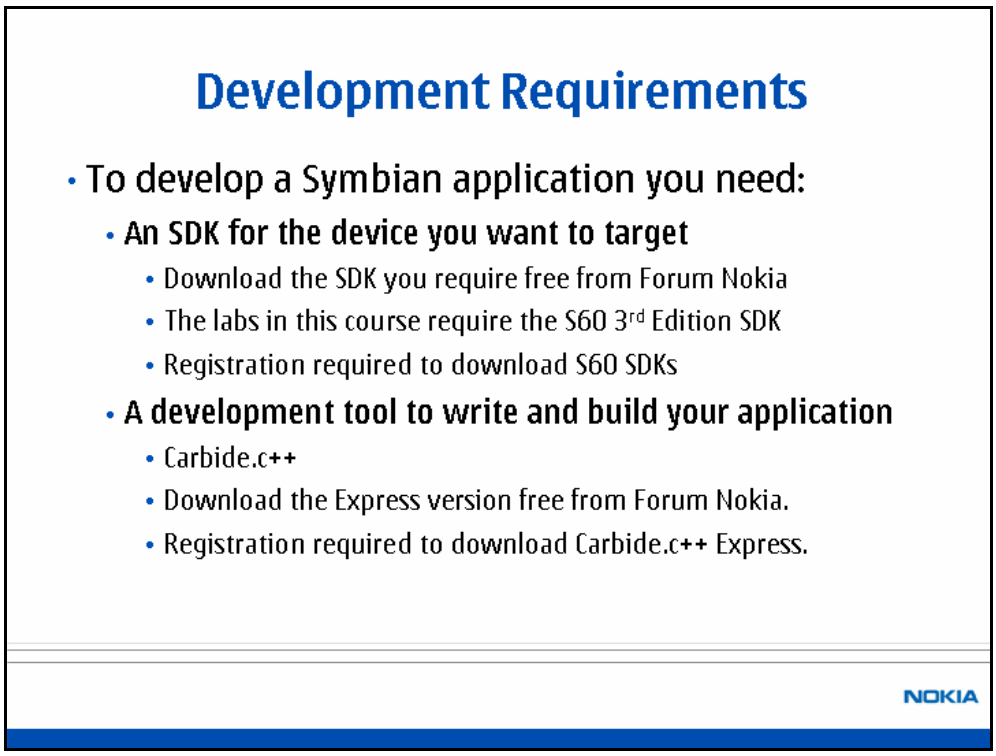
The Symbian OS has undergone rapid and extensive development. The version of Symbian OS used in the UI design platforms (discussed on previous slides) is shown in the table on the above slide and discussed below:

- Symbian OS v6.0 - This is the version of Symbian OS used in early Series 80 devices (Nokia 9210).
- Symbian OS v6.1 - This is the version of Symbian OS used in S60 1st Edition devices. It added support for GPRS, Bluetooth and MMS message on top of previous functionality. This version of Symbian OS is also used in FOMA devices.
- Symbian OS v7.0 - This is the version of Symbian OS used by UIQ v2,0 and 2.1 devices; e.g. Sony Ericsson P800. It added 3G support and improved communications features such as introducing USB support and HTTP APIs. This version of Symbian OS is also used in the Nokia 7710.
- Symbian OS v7.0s – This is the version of Symbian OS used in S60 2nd Edition. Only the Nokia 6600 was produced using this version of the platform. It added further new telephony features and a new multi-media framework amongst other enhancements.
- Symbian OS v7.0s (enhanced) – This is the version of Symbian OS used in S60 2nd Edition Feature Pack 1 (FP1) and is used in devices like the Nokia 7610.
- Symbian OS v8.0 – This is used in S60 2nd Edition Feature Pack 2 (FP2) devices. Key developer features are improved methods of IPC, new Multimedia extensions, and new MIDP APIs.
- Symbian OS v8.1 - This is the version of Symbian OS used in S60 2nd Edition Feature Pack 3 (FP3) and is used in devices like the Nokia N70.
- Symbian OS v9.1 - This is the version of Symbian OS used in S60 3rd Edition and is used in devices like the Nokia 3250. It is also used in UIQ 3.0 where it is used in

devices like the Sony Ericsson M600i. The key features of this new version are platform security and data caging as well as the introduction of a new kernel.

- Symbian OS v9.2 - This is the version of Symbian OS used in S60 3rd Edition Feature Pack 1. It adds support for Bluetooth 2.0 and OMA Device Management 1.2. It is used in devices like the Nokia N76.
- Symbian OS v9.3 – This version provides updated communication and multi-media functinality.

Development Requirements



UI platforms, intended to support the installation of third-party applications written in native C++, have to be supported by an SDK which defines that UI platform (or at least a particular version of it). Use the following link as a starting point in navigating to and downloading the SDK you want:

http://www.symbian.com/developer/sdks/index.asp

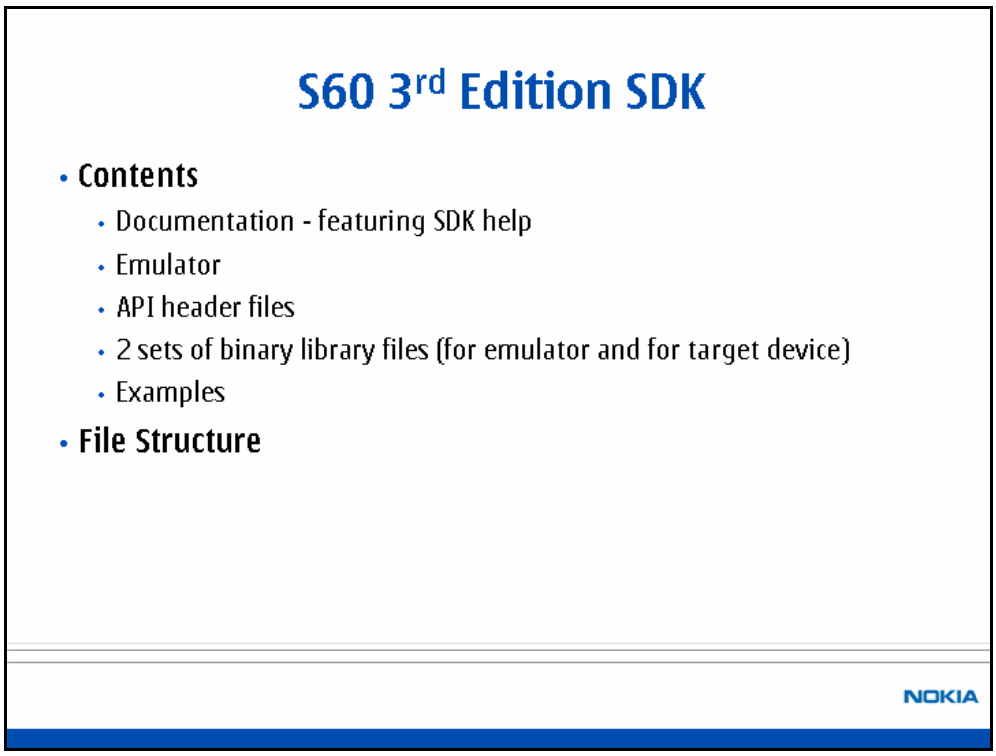
For the labs in this course you will need the S60 3rd Edition SDK. You should first download the SDK. Note that this is over 200 MB in size so will take a while to download. Once you have downloaded it, you should unzip and install it following the instructions contained in the installation guide (which is a pdf file that is part of the SDK).

As well as an SDK, you will also require a development tool to write and build your application. Various tools are available; e.g. Codewarrior, Carbide.c++, Microsoft Visual Studio etc but note that some tools will only work with particular SDKs. Consult the SDK documentation to find out which tools your SDK will work with.

Since we shall be working with the S60 3rd Edition SDK the best tool to use is Carbide.c++ Express which is a free tool available for download from the Forum Nokia website. Note, however, that this tool is for personal use only and cannot be used commercially.

Also note that to download any of the S60 SDKs or Carbide.c++ Express you must be registered with Forum Nokia. Registration is free; you just have to fill out some details in an online form.

S60 3rd Edition SDK

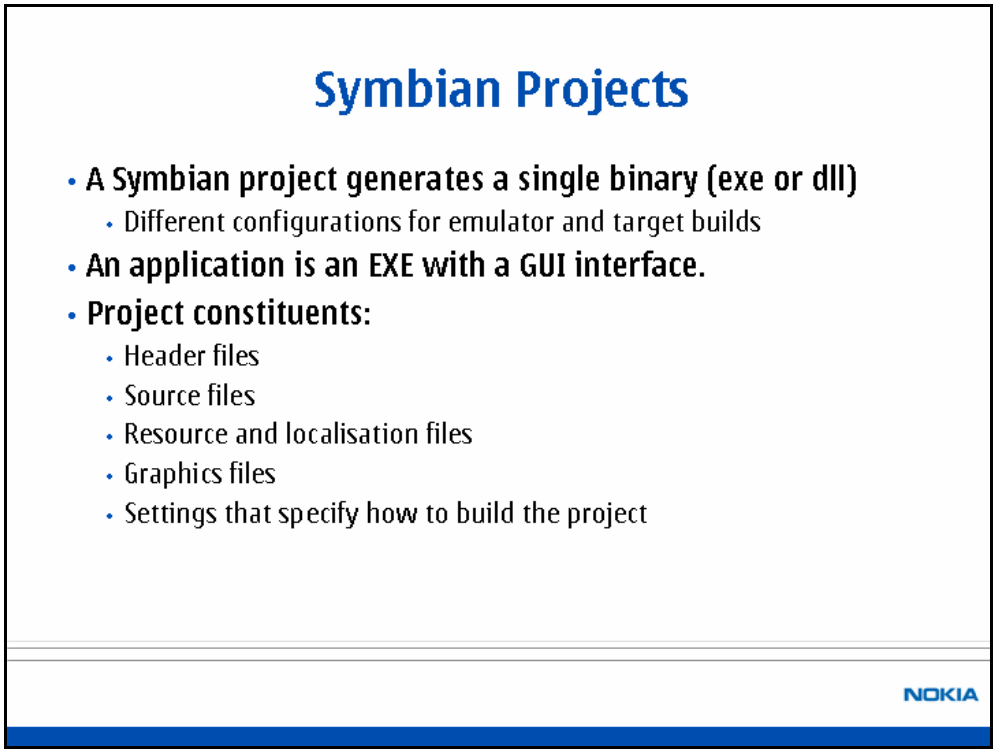


Some useful information about the S60 3rd Edition SDK is presented below.

- The SDK contains a significant amount of documentation on various subjects. This may be found in the `<EPOCROOT>\S60Doc` folder where `<EPOCROOT>` is the root folder location of your SDK. If you installed your SDK to the default location then `<EPOCROOT>` is `C:\Symbian\9.1\S60_3rd`.
- Amongst the documentation is a help reference, called `S60_CPP_SDK_3rd.chm`, that contains a S60 API reference. On Windows XP this may also be accessed by activating the Start menu and selecting the "All Programs -> S60 Developer Tools -> 3rd Edition SDK -> 1.0 -> SDK Help" menu item.
- The SDK contains an emulator The emulator is a port of the S60 platform to the Win32 platform. For most circumstances, this provides a faithful emulation of application behaviour on a pc. The emulator runs in a single Windows process called epoc.exe. To launch the emulator activate the Start menu and select the "All Programs -> S60 Developer Tools -> 3rd Edition SDK -> 1.0 -> Emulator" menu item.
- The SDK includes all the necessary header files to use the S60 API. These are found in the `<EPOCROOT>\Epoc32\include` folder.
- The SDK contains two sets of library files used by the linker during the latter stages of the build process. The first set is used in building applications that will run on the emulator. These have a .lib extension and are located in the `<EPOCROOT>\Epoc32\release\winscw\udeb` folder. The second set of library files is used in building applications that will run on an S60 3rd Edition device, for example the Nokia 3250. These usually have a .dso extension and are located in the `<EPOCROOT>\Epoc32\release\armv5` folder.

- The SDK also comes with a number of different code examples that can be used to find out how to do specific tasks in Symbian OS. These are located in the `<EPOCROOT>\Examples` folder.

Symbian Projects



A Symbian/S60 project encapsulates all the resources required to generate a single binary file, whether it be an EXE or a DLL. Different configurations are used within a project; one to build for the emulator and another to build for an S60 3rd Edition device.

An application is a binary EXE file that implements a GUI interface.

The constituents of a Symbian OS/S60 project are described below:

- Header files – Specify definitions that are included in source files.
- Source files – Compiled by a compiler into object code, which is then linked to generate an EXE or DLL.
- Resource files – Text files that define resources (for example strings and menus) that are built as part of the build process into binary resource files.
- Localisation files – Text files that define specific strings in an application that are language dependent. Instead of hardcoding the strings in the application they are defined in this file and only referenced in the application.
- Graphics Files – Symbian OS/S60 support the use of bitmaps and Scalable vector graphics. Files of a particular type are combined into a single file; a multi-bitmap file file for bitmaps and a multi-icon file for SVGs.
- Settings – Either contained in IDE specific settings files or contained in a project's MMP file. An example of an important setting is the list of libraries to link against.

Developer
Training

Module #04302

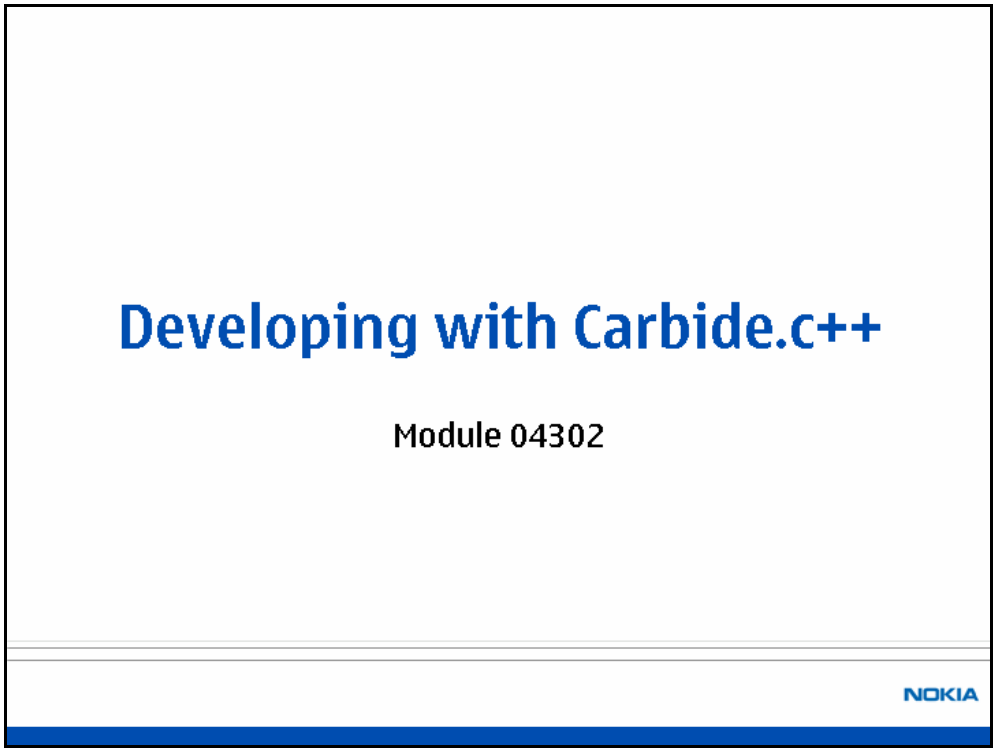
Developing with Carbide.c++

Contents

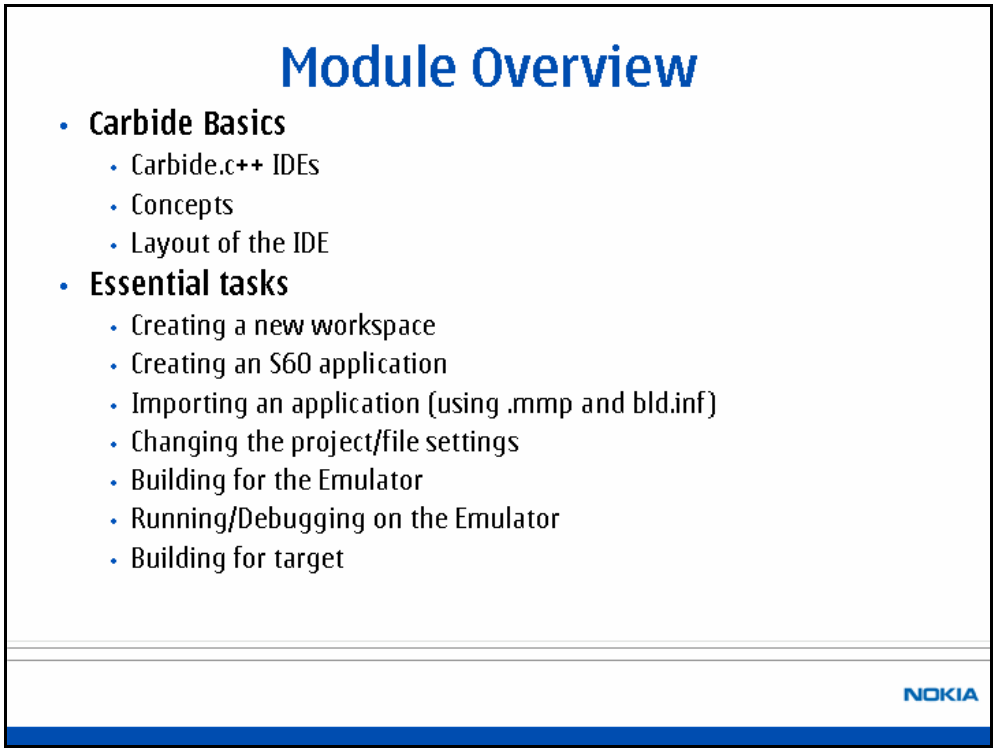
NOKIA

BO-EN-04300-20080108

Developing with Carbide.c++

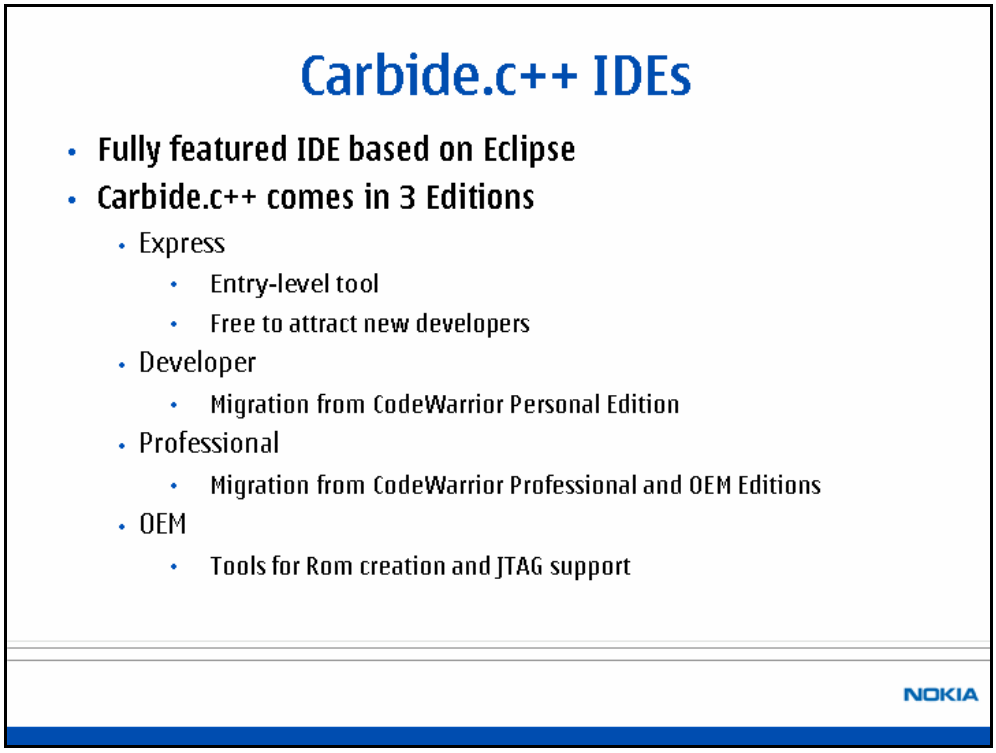


Module Overview



This module presents an overview of Carbide.c++ and illustrates how to perform the most essential of development tasks with this IDE.

Carbide.c++ IDEs



The new Carbide brand of developer tools provides a single identity for all Nokia's developer tool offerings. Carbide.c++ is the natural evolution of Nokia's CodeWarrior for Symbian OS suite of development tools, providing a modern IDE with excellent usability mated to powerful Symbian OS debugging created by Nokia's Symbian OS development tools team.

Carbide.c++ is developed in cooperation with Symbian and built on Eclipse; the rapidly evolving open-source integrated development environment framework designed to encourage software tool innovation.

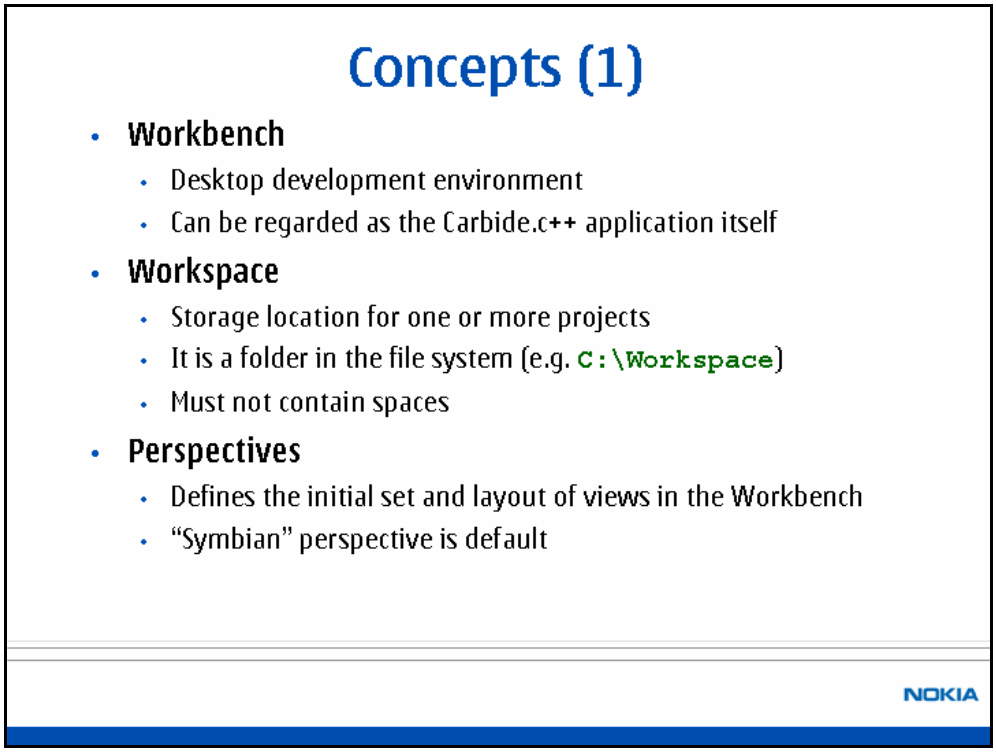
Carbide.c++ comes in 4 editions:

- Carbide.c++ Express – This is an entry-level tool available for download at no cost and designed to invite all programmers to explore Symbian OS, providing a complete set of application development tools needed to target the S60 SDKs and to build and deploy applications to devices.
- Carbide.c++ Developer Edition - This IDE offers a new graphical Rapid Application Development (RAD) tool to ease C++ development for S60 devices. It meets the requirements of advanced-level application developers by providing improved productivity tools for application development and on-target debugging on S60 3rd Edition devices.
- Carbide.c++ Professional Edition – This product is designed for professional Symbian OS developers who focus on Symbian device development and high performance applications and who require early access to next-generation Symbian OS support and technical specifications.
- Carbide.c++ OEM Edition – This product is designed for OEMs to create solutions for full Symbian device development.

All CodeWarrior for Symbian OS users have a clear and consistent migration to Carbide. Advanced developers working with CodeWarrior Personal Edition will migrate to Carbide.c++ Developer Edition. CodeWarrior Professional and OEM Edition developers will easily migrate to Carbide.c++ Professional. Developers working with CodeWarrior OEM Edition's JTAG debugging will be offered an optional OEM plug-in to the Carbide.c++ Professional Edition, maximizing the flexibility for customers to acquire and configure Carbide tools according to their needs.

Carbide.c++ Express is designed to attract new developers to the Symbian OS community via the popular Eclipse IDE bundled with no-cost C++ Symbian OS development tools, lowering many barriers for large numbers of novice developers, students, academics, and evaluators to begin developing on Symbian OS.

Concepts (1)



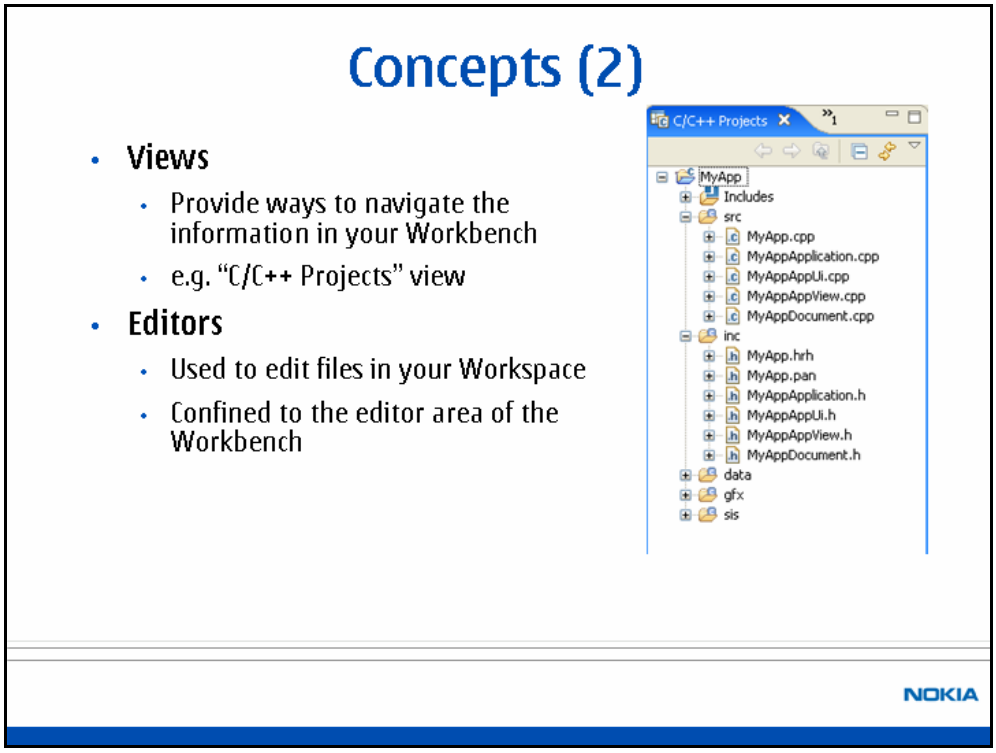
The common concepts associated with Carbide/Eclipse are discussed below.

In the Carbide.c++ Help documentation, the Workbench is referred to as the desktop development environment. For the purposes of this course it can be regarded as the Carbide.c++ application itself.

The projects, folders and files that you create with the Workbench are all stored under a single directory that represents your Workspace.

Each Workbench window contains one or more perspectives. Perspectives define the initial set and layout of views in the Workbench and control what appears in certain menus and tool bars. More than one Workbench window can exist on the desktop at any given time. The default perspective is "Symbian" and should be used for all development work.

Concepts (2)

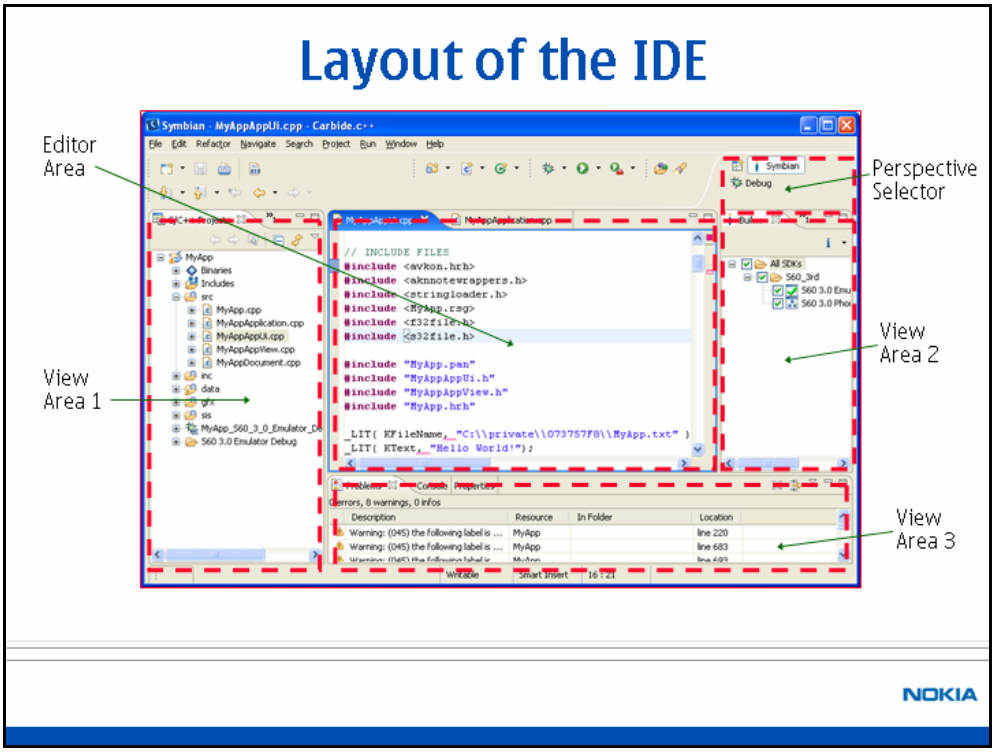


Views are UI components that provide ways to navigate the information in your Workbench. For example, the "C/C++ Project" view displays the projects located in the current workspace using a tree structure. Top level nodes are 'project' nodes that can contain child 'folder' nodes. These nodes themselves can contain child 'file' nodes.

Editors are used to edit files in your workspace. They are another UI component, but unlike Views are associated with a particular file and are confined to the editor area of the Workbench. Editors identify the file they are modifying by a tab containing the name of the file.

See the Carbide.c++ Help documentation for more information on any of these topics including specific information about each of the views. Select the "Help -> Help Contents" menu item to open the help window. Once this window is open, type a search string in the top left search field and click the Go button.

Layout of the IDE



The above slide illustrates the default layout of the Carbide.c++ IDE (for the Symbian perspective).

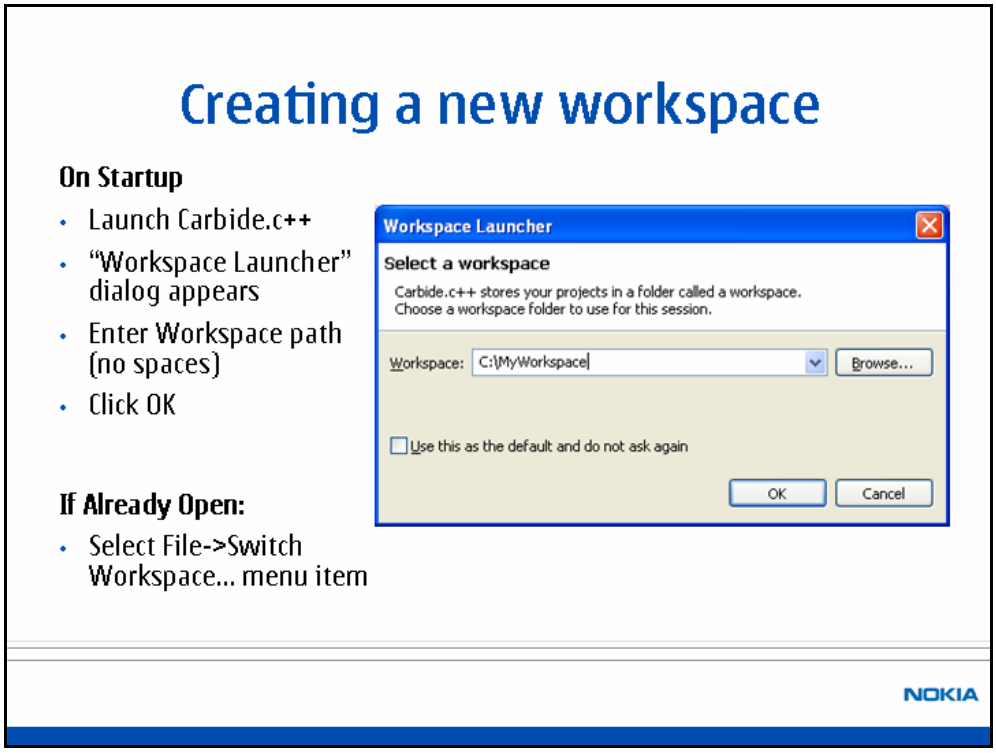
In the top right corner of the IDE a control exists to display the current perspective and allow the selection of other perspectives.

The editor area appears in the centre of the IDE and is surrounded by view windows (although in the slide it is only surrounded on three sides).

The default view layout is shown in the above slide with three view areas. However, in practise you can have any number of view areas. View windows may be moved from one view area to another or even into a completely new view area by dragging it's title bar with the mouse into another area of the IDE. Views may also be detached rather than appearing in a specific view area. In such cases the view has its own window which is independent from the main Carbide.c++ IDE window.

To reset the initial layout of your current perspective select the "Window -> Reset Perspective" menu item.

Creating a new workspace



The first task in using Carbide.c++ is to select a workspace. As mentioned before, your workspace is a single directory under which all the projects, folders and files that you create with Carbide.c++ are all stored.

When you open Carbide.c++ for the first time you will be presented with the "Workspace Launcher" dialog. You can choose the workspace folder in one of three ways:

- Type name of the folder (including its path) into the Workspace field.
- Select the Browse button and browse to the folder of your choosing.
- Select (an existing) workspace from the drop down menu.

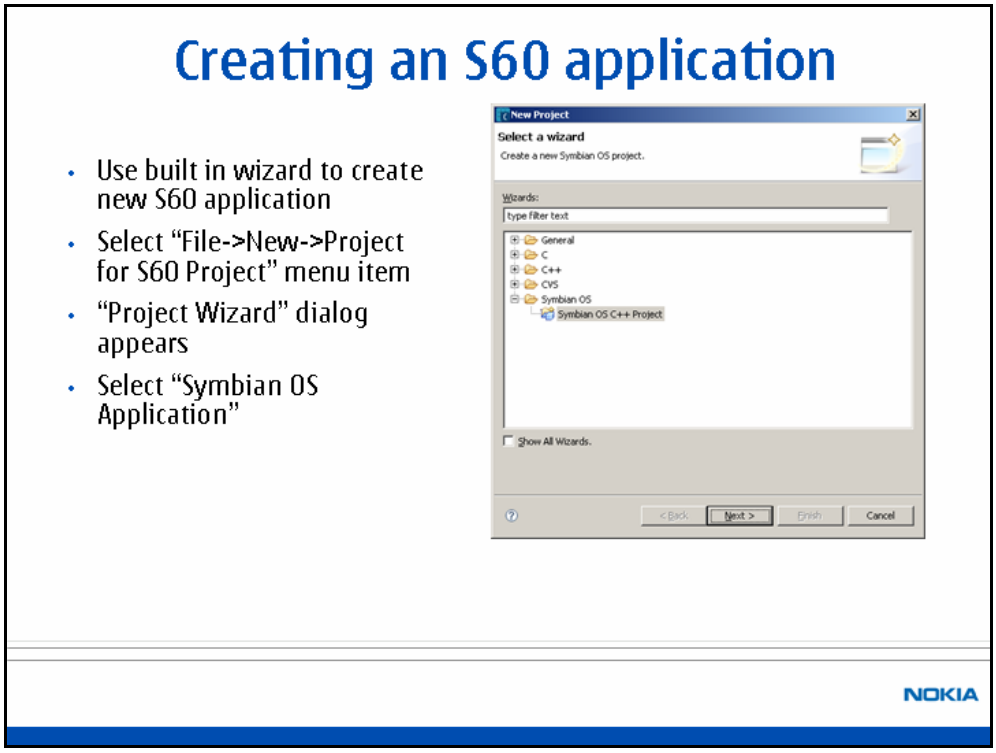
A new workspace is simply a folder that has not previously been used as a Carbide.c++ workspace and can be selected by either of the first two methods.

An existing workspace is a folder that has previously been used as a Carbide.c++ workspace and can be selected by any of the above methods.

Note that the workspace folder (including its path) must not contain any spaces. Once you are happy with your choice click the OK button. After a brief initialisation period the Carbide.c++ window will be displayed.

You may switch between different workspaces at any time using the "File -> Switch Workspace..." menu item.

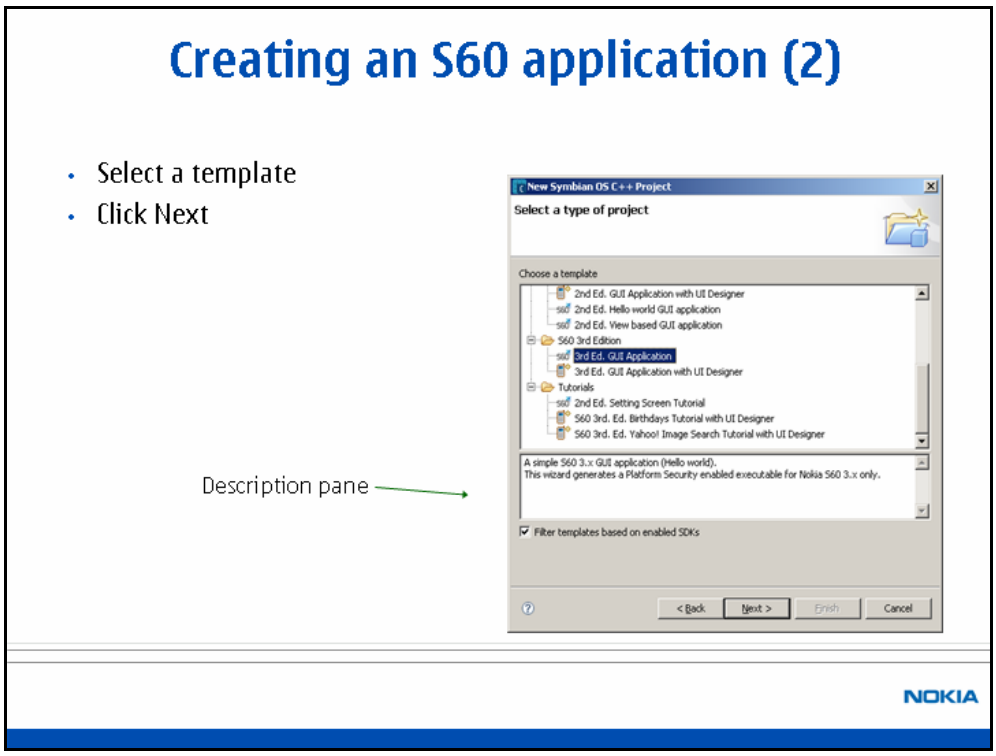
Creating an S60 application



To create a new Carbide.c++ project, one of the application creation wizards should be used. These wizards generate the code, and settings, for a simple application, that can be used as a starting point for writing your own application.

This slide, and the following ones, illustrate how to create a new Carbide.c++ project using the "New C++ Application for S60 Project" wizard.

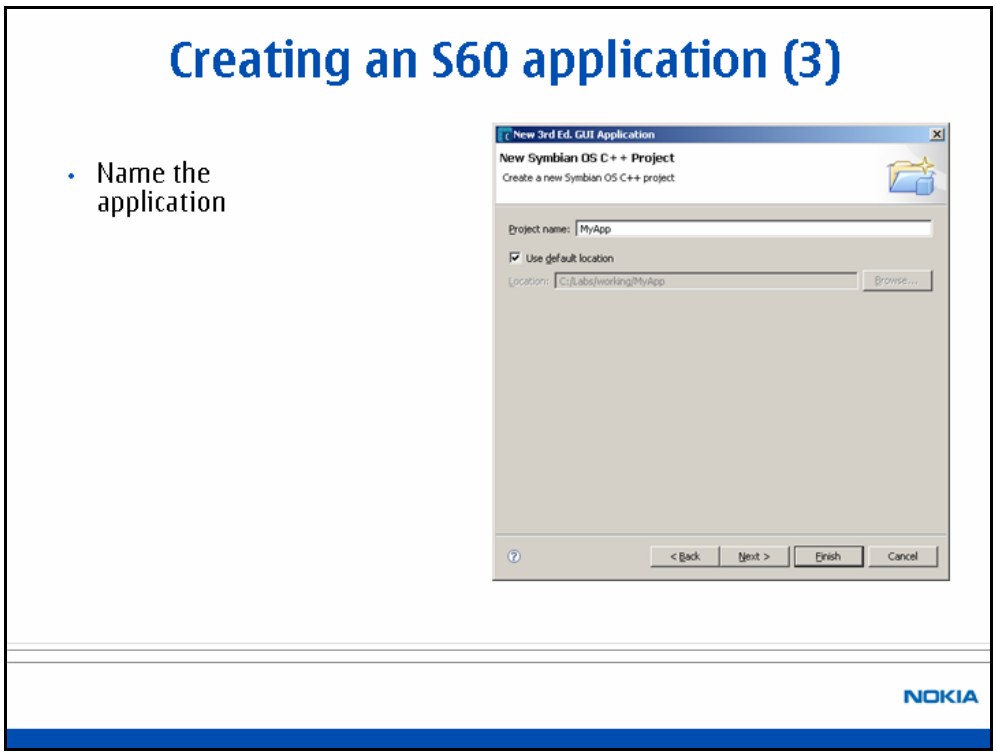
Creating an S60 application (2)



The next screen of the wizard allows you to choose a template for your application. A simple explanation is given in the description pane (see slide) for the template currently selected in the list.

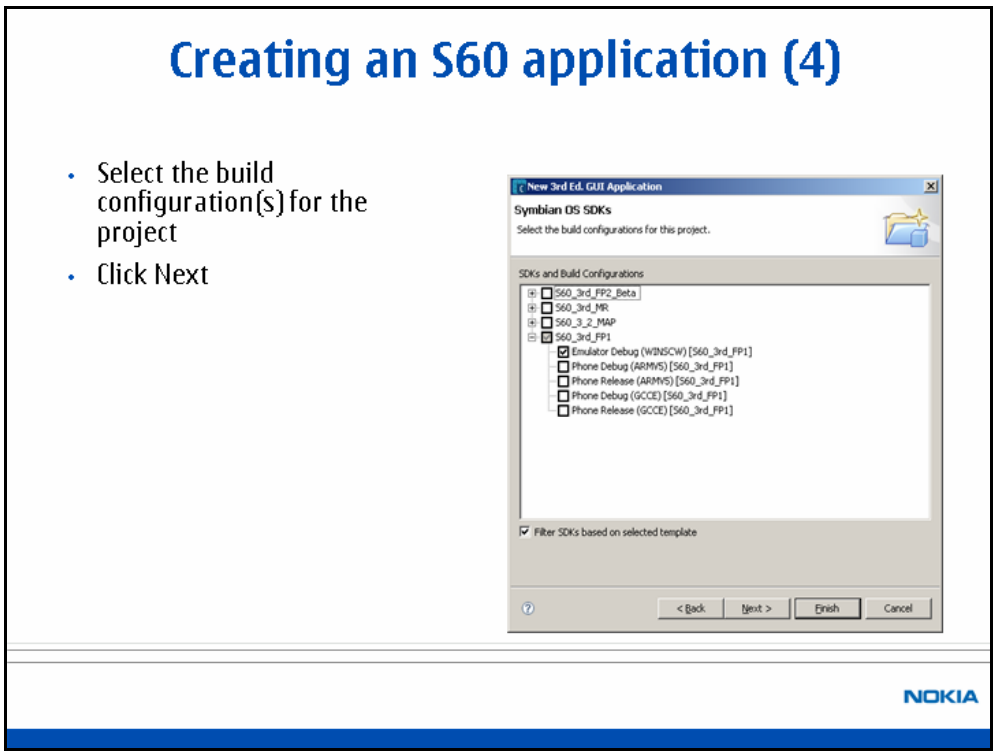
To create a project for S60 3rd Edition, select the "3rd Edition GUI Application" template and click the Next button.

Creating an S60 application (3)



The next screen of the wizard allows you to choose a name for your application.

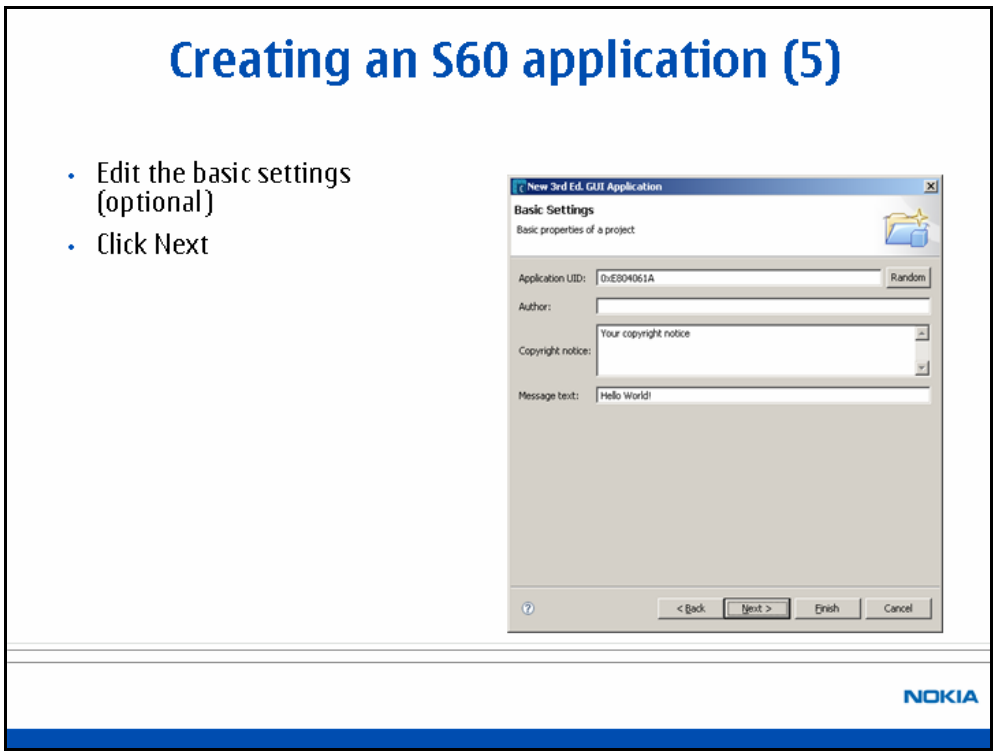
Creating an S60 application (4)



The next screen of the wizard allows you to select the build configurations you require for your project. The list of build configurations you can choose from depends on the project template you chose and the SDKs you have installed. Note that configurations are grouped together on a per SDK basis. An SDK will usually support one debug configuration for the emulator and one or two release configurations for target phones.

To select or unselect a configuration click the checkbox next to the name of the configuration. A green arrow inside the checkbox indicates the configuration is selected.

Creating an S60 application (5)



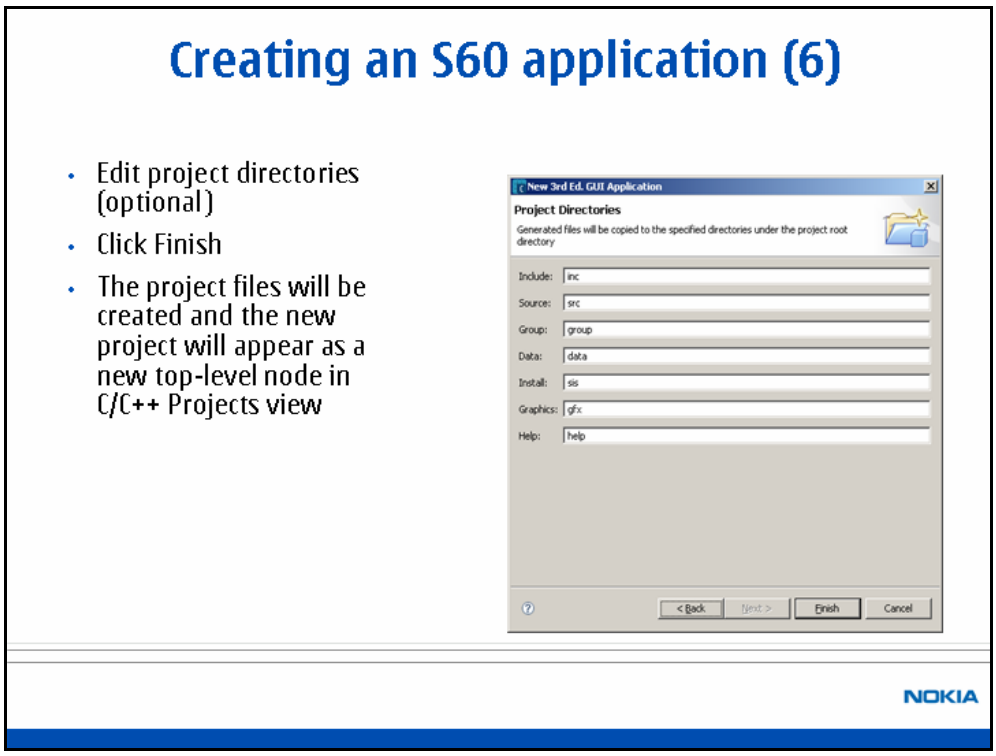
The next screen of the wizard allows you to set some of the basic settings your project will have. Each of these is explained below:

- The Application UID is a 32 bit identifier that uniquely identifies your application from all other applications on the system. You should not edit this value!
- The Author and Copyright notice are used in file header comments in source code files.
- The “Text to be shown” setting is used in the code as the text displayed in an information note. The note is displayed when a menu item is selected.

Warning: Do not type a string longer than 32 characters for the “Text to be shown” setting, otherwise your project will not build. This is due to the code generated by the wizard generated code” being implemented in way so that is doesn’t deal with long strings.

Edit the settings as desired and click the Next button. Alternatively you may click the Finish button. This bypasses the following screens and starts the process to create your new project.

Creating an S60 application (6)

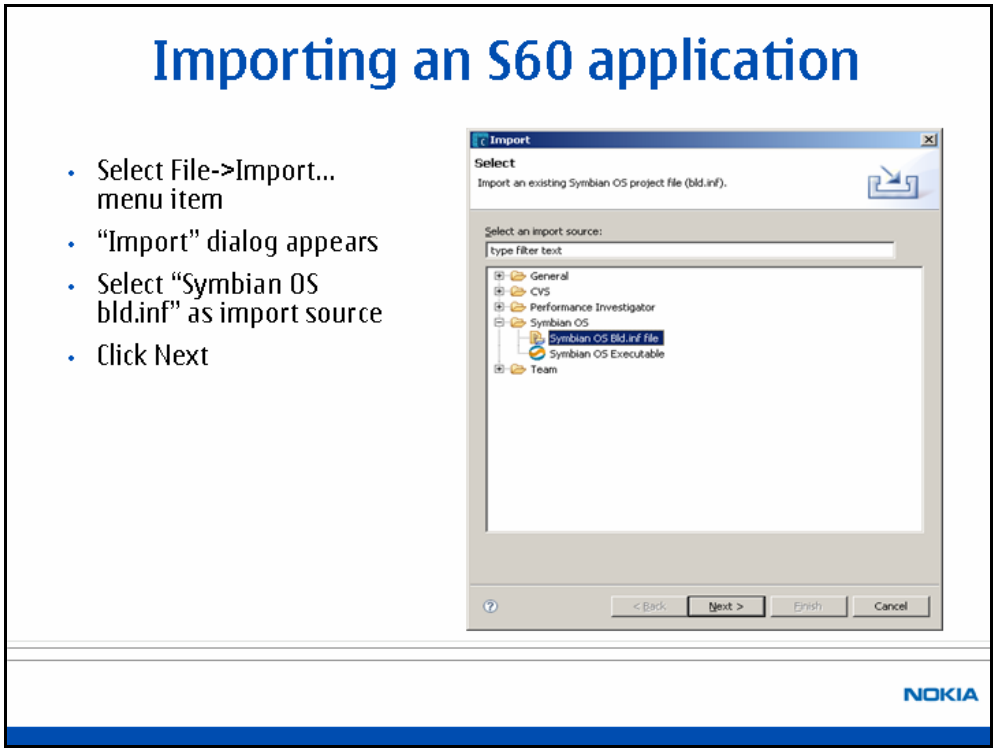


The next screen of the wizard allows you to set the names of the project directories your project will be created with. The structure is designed to hold different types of file in different directories as explained below:

- The Include folder contains C++ header files.
- The Source folder contains C++ source files.
- The Data folder contains Symbian OS resource files.
- The Graphics folder contains SVGs and bitmaps.
- The Install folder contains installation related files.

Edit the directory names if you wish and select the Finish button. The wizard starts creating your new project. Once it has finished, a new top-level 'project' node, with the name of the project, appears in the "C/C++ Project" view.

Importing an S60 application

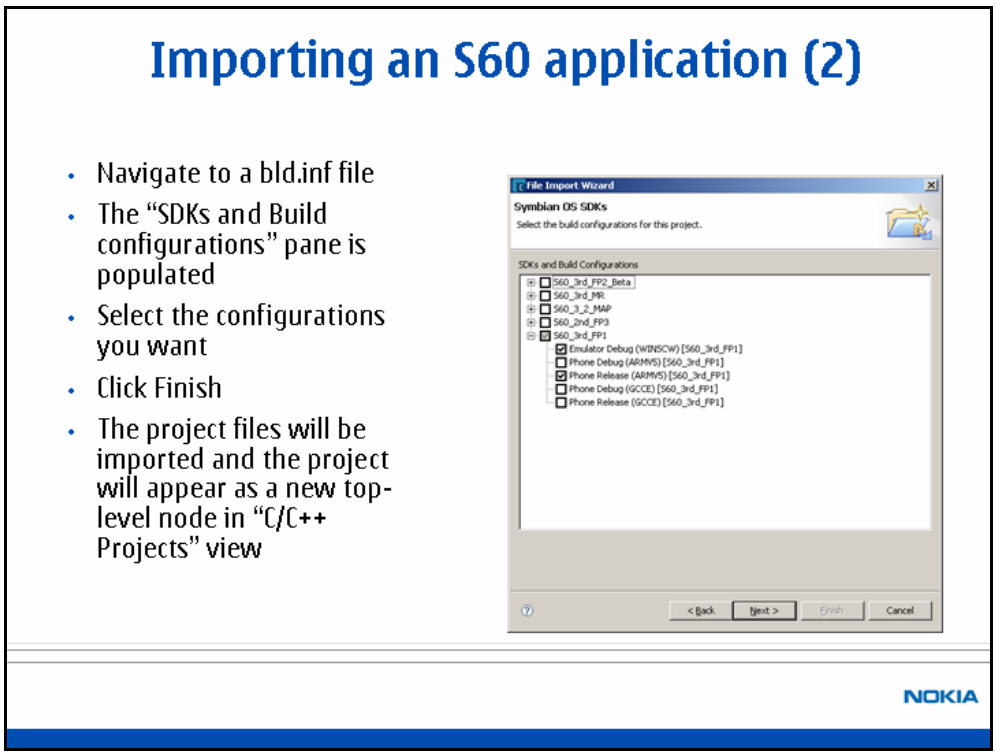


Rather than create a new project, you may want to import an existing project (with an MMP file specification) into your Carbide.c++ workspace. These slides show you how to do this.

Select the "File -> Import..." menu item to display the Import dialog. From the list of import sources, choose "Symbian OS bld.inf file" and click the Next button.

Each bld.inf file can contain references to multiple mmp files.

Importing an S60 application (2)



The screen changes to allow the selection of the bld.inf file associated with the project(s) you want to import. Either type the name (including path) in the "MMP file" field or browse to it using the Browse button.

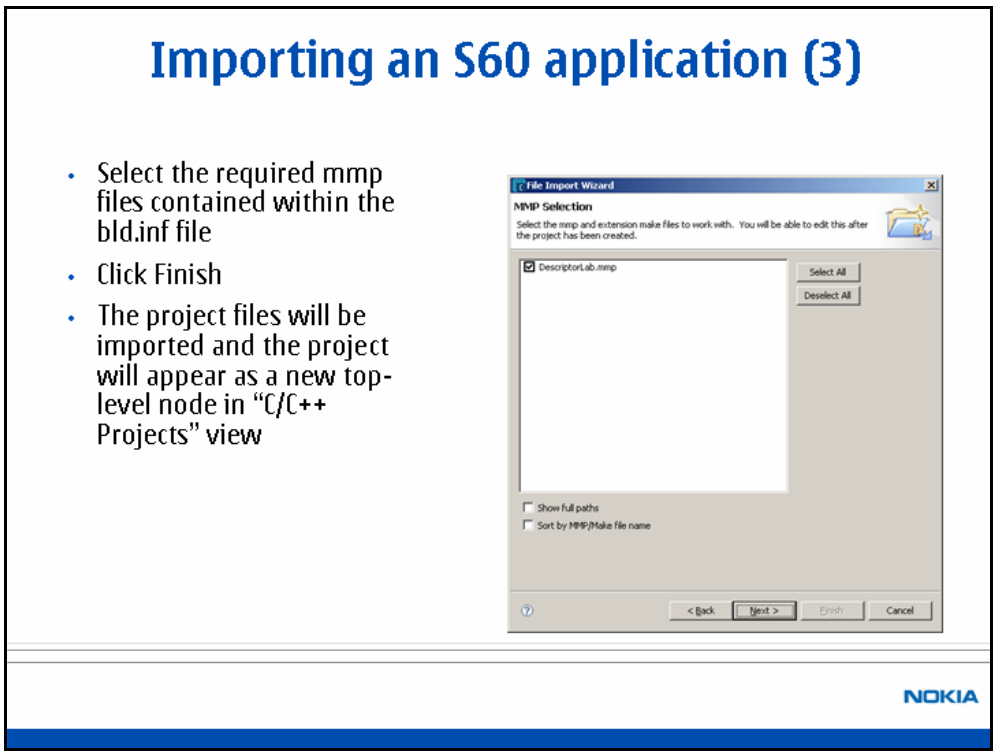
When a valid file name is entered the "SDKs and Build configurations" pane is populated; i.e. a list of possible configurations to choose from is displayed. This list depends on the MMP file you chose and the SDKs you have installed. Note that configurations are grouped together on a per SDK basis. An SDK will usually support one debug configuration for the emulator and one or two release configurations for target phones.

To select or unselect a configuration, click the checkbox next to the name of the configuration. A green arrow inside the checkbox indicates the configuration is selected.

You must select the correct configurations for the project being imported. If the project you are importing is an S60 3rd Edition project select the S60_3rd configurations only.

Click Finish to initiate the import. Once it has finished a new top-level 'project' node, with the name of the imported project, appears in the "C/C++ Projects" view.

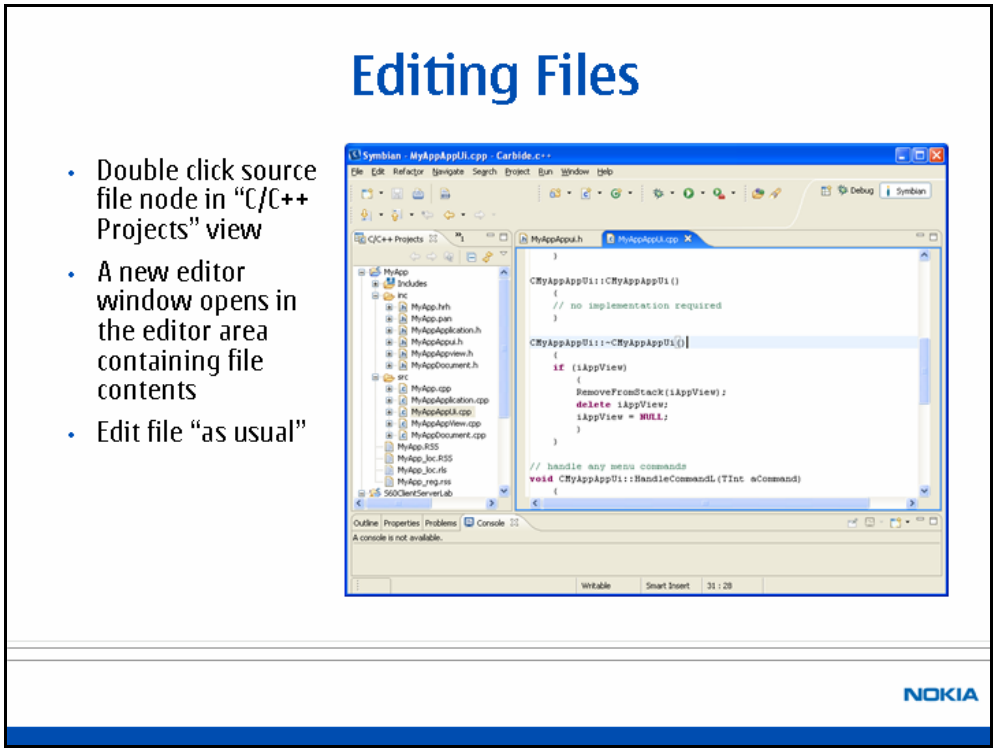
Importing an S60 application (3)



The screen changes to allow the selection of mmp files from within the bld.inf previously selected.

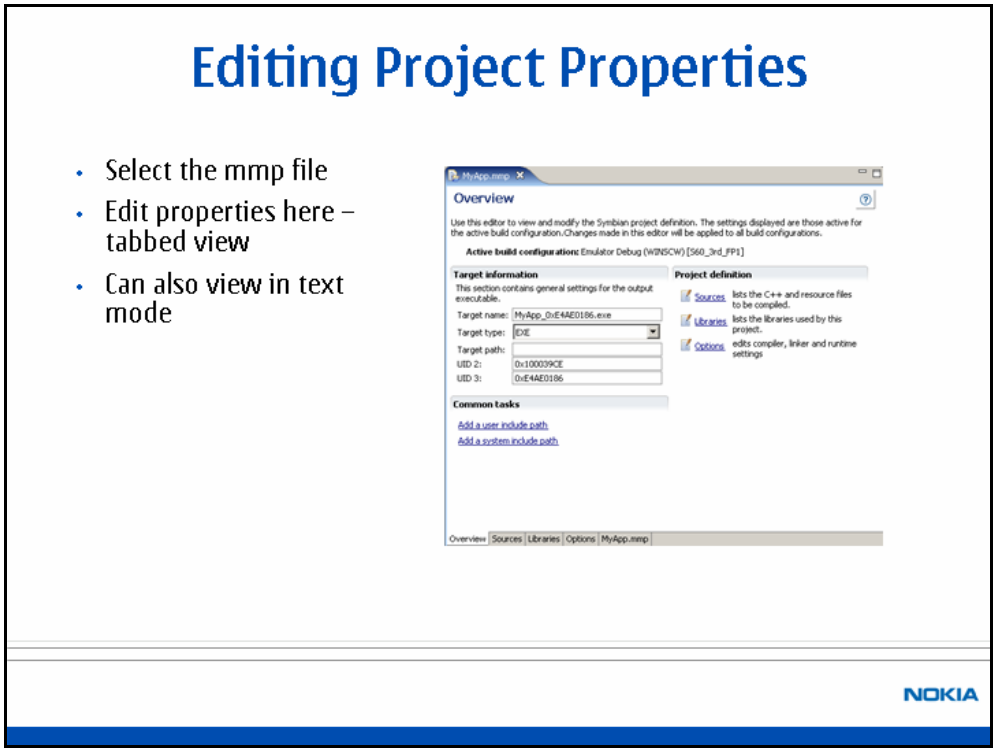
Only mmp file selected here will be imported.

Editing Files



Editing of files in Carbide.c++ is very straightforward. In the "C/C++ Projects" view, double click on a source file node that you want to edit. A new editor window opens in the editor area containing the file contents. Edit the file as "usual"; i.e. as you would using any other standard text editor.

Editing Project Properties



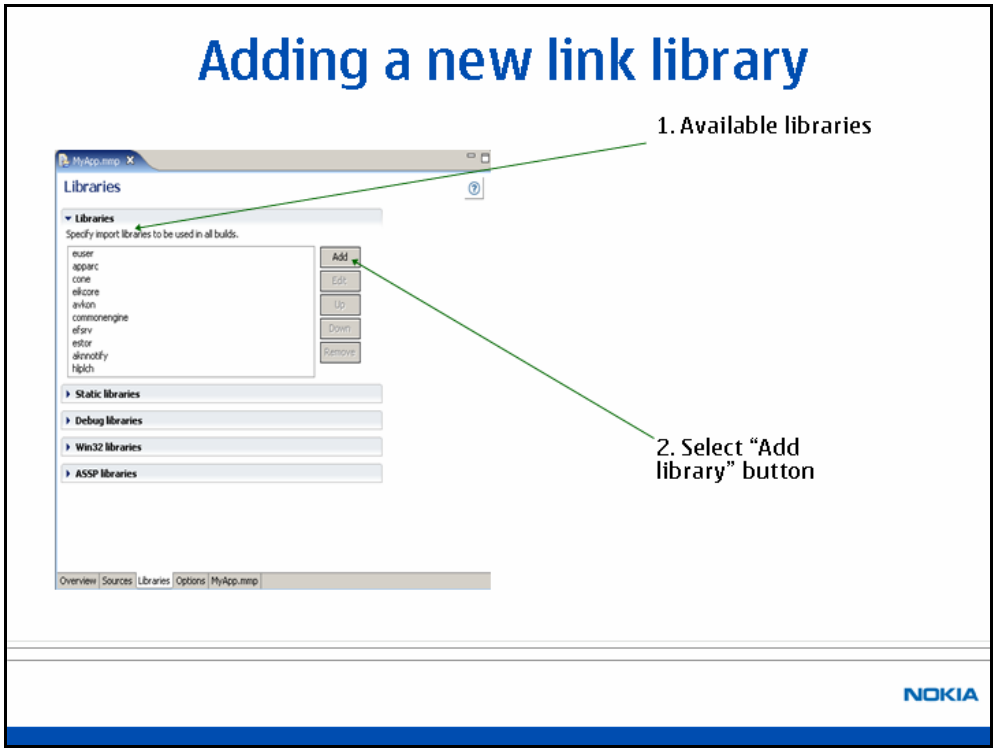
The properties of a project are specified in a special type of text file called an MMP file. However, when using Carbide.c++, there is no need to edit the file as text all the settings needed for your project are managed for you by Carbide.c++. Carbide provides a project properties dialog to enable you to edit these settings in an easy manner.

To display the project properties select the mmp file from the folder view.

Carbide shows a dialog that allows you to edit properties in the mmp file, there are tabs along the bottom of the dialog.

It is alos possible to view and edit the mmp file as plain text.

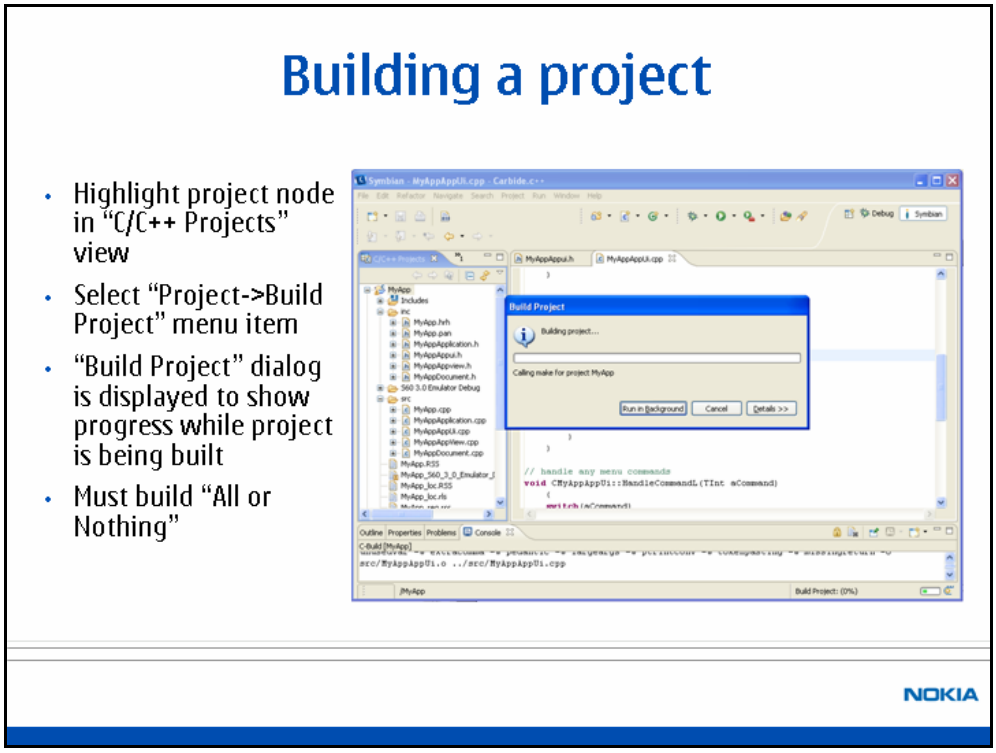
Adding a new link library



The most frequent change to the settings will, most probably, be adding link libraries for your application.

Libraries can be added from the mmp file view "Libraries" tab.

Building a project



To build an application in Carbide.c++ is a simple task. In "C/C++ Projects" view, simply highlight the project node that you wish to build and select the "Project ->Build Project" menu item. Once selected the build begins. Alternatively, you may right click on a project node to display a context menu and select the "Build Project" menu item.

Note that Carbide.c++ makes no provision for building individual parts of a project. You have to build everything in one go.

During the build, a "Build Project" dialog is displayed to show the progress of the build. When the build completes this dialog disappears. Note that you may run the build in the background (so that you can continue working while it completes) by selecting the "Run in Background" button on the dialog. If this button is pressed the dialog disappears (during the build) and build progress is indicated on the status bar.

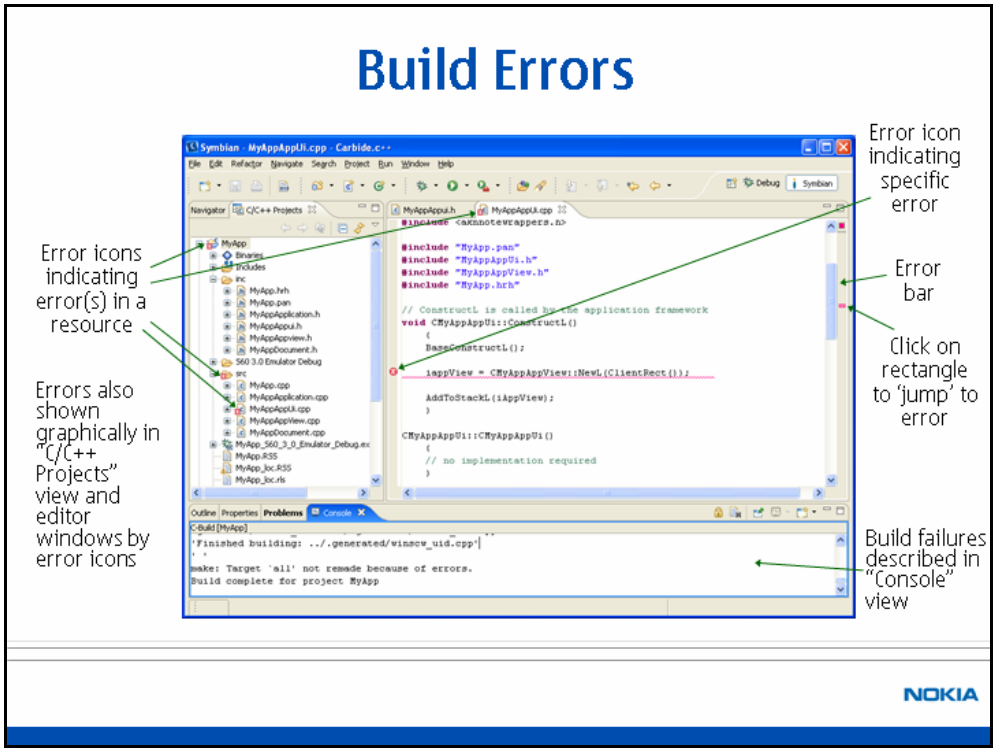
A log of the build is output to the Console view. This should be checked after the build has completed to see if the build completed successfully or not. If the penultimate line of output text in the Console view is:

```
make: Target `all' not remade because of errors.
```

then one or more errors are present in your project. You must fix them and build your project again. The next slide discusses how to locate errors in your project.

After the initial use of the "Build Project" menu item to build your project, selecting it again will only build those parts of the project that have been modified since the last build took place. To rebuild the whole project at any time use the "Project -> Rebuild Project" menu item instead.

Build Errors



If you have build failures in your project then you must fix them. The errors will all be described in the build log that was output to the Console view during the build. This will tell you the file and line number the error occurred on along with a reason for the error.

Errors are also indicated, and can be located, using the graphical error icons. These are used to indicate whether an error is present in a particular resource.

To show how to track an error, using this graphical approach, let us look at the example on the slide. First look at the "MyApp" project node in the "C/C++ Projects" view. This has an error icon associated with it so there are error(s) in the project.

Expand the "MyApp" project node to view its child nodes. At least one of these nodes will have an associated error icon indicating there are error(s) associated with that resource. The "src" folder node is the resource with the associated error icon and so contains the error(s).

Expand the "src" folder node to view its child nodes. Like before at least one of these nodes will have an associated error icon indicating there are error(s) associated with that resource. In this case, it is the MyAppAppUi.cpp source-file node that contains the error(s).

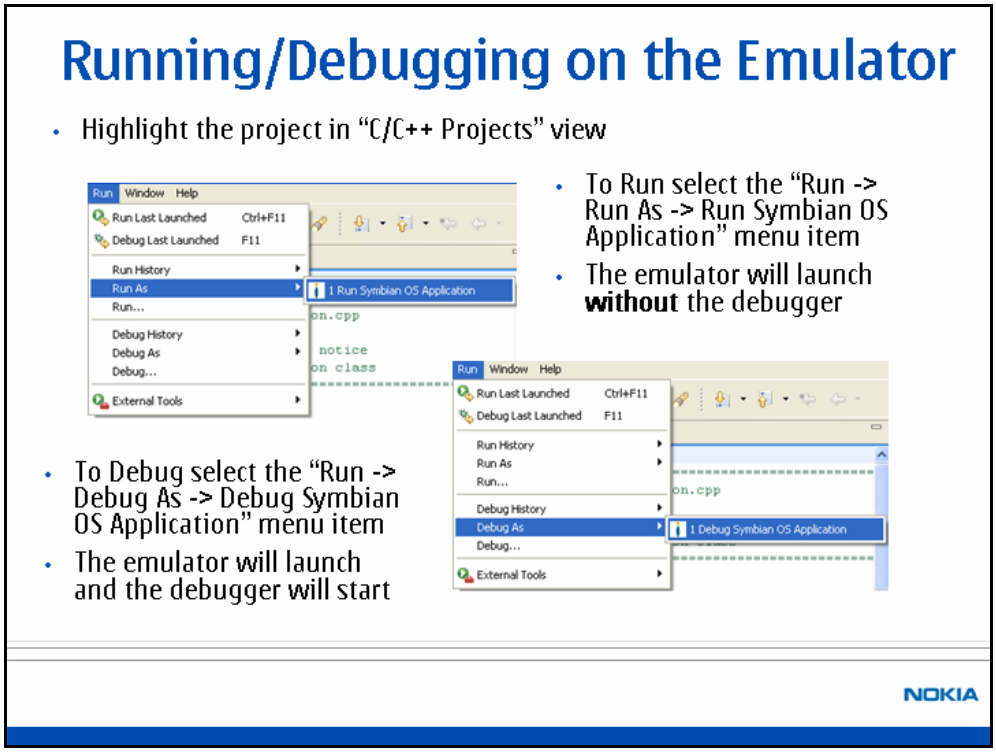
You can repeat this procedure and expand the MyAppAppUi.cpp source-file node to find which function(s) the error(s) occurred in. In any event, the next stage is to open the source file MyAppAppUi.cpp by double clicking on the source-file node or a function node with an associated error icon. This will open an editor window containing the contents of the file.

The error bar is the area immediately to the right of an editor window's vertical scroll bar. It works in conjunction with the scroll bar to show the location of all the errors in the file. Each error is represented by a red rectangle. Clicking on a rectangle causes the editor window to jump to the part of the file where the error is located.

The editor window also displays an error icon in the marker bar for each line of the file on which an error occurs. The marker bar is located to the left hand side of the editor window. In addition the line itself on which the error occurs is underlined in red.

If you move the mouse pointer over an error icon in the marker bar or an error rectangle in the error bar, the reason for the error (as described in the build log) will be shown as a tooltip.

Running/Debugging on the Emulator

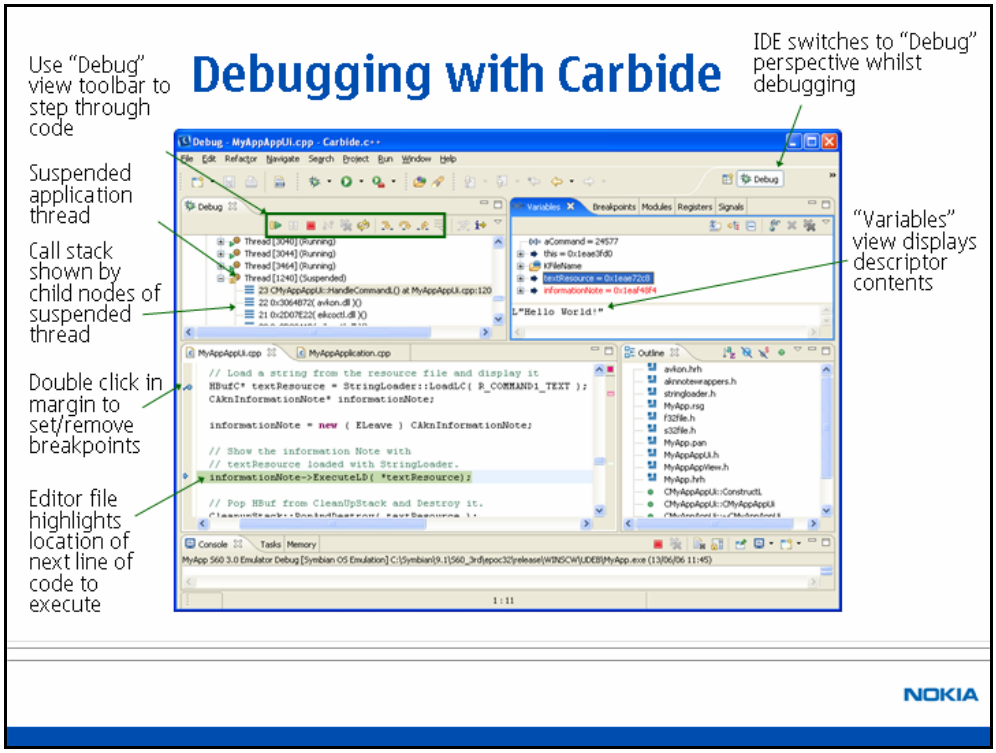


Once you have fixed all your errors and built your project successfully you can now test your application out. The simplest way to test your application is to use the emulator that comes with an SDK. There are two choices available:

1. Run the application in the emulator. Select the "Run -> Run As -> Run Symbian OS Application" menu item. The emulator will launch without the debugger.

2. Debug the application in the emulator. select the "Run -> Debug As -> Debug Symbian OS Application" menu item. The emulator will launch and the debugger will start.

Debugging with Carbide



Once the debugger has started, Carbide.c++ automatically switches to the Debug perspective. This perspective displays views that are useful when debugging an application.

In order to debug code you must place a breakpoint somewhere in your code. This is done by double clicking in the marker bar (to the immediate left of the editor window) alongside the line of code where you want to place the breakpoint. A blue circle appears to indicate that a breakpoint has been set. To remove a breakpoint double click on the breakpoint icon in the marker bar.

When the breakpoint you set has been reached by the execution flow, the debugger will halt execution of your application and focus will switch from the emulator to the Carbide.c++ IDE. The line of code that is next in line to be executed is highlighted in green in the editor window.

When the debugger halted execution of your application, it suspended your application's thread of execution. This is shown in the Debug view which contains a list of all the Symbian threads running under the emulator. One of the thread nodes in the list is marked "Suspended"; this is your application's thread.

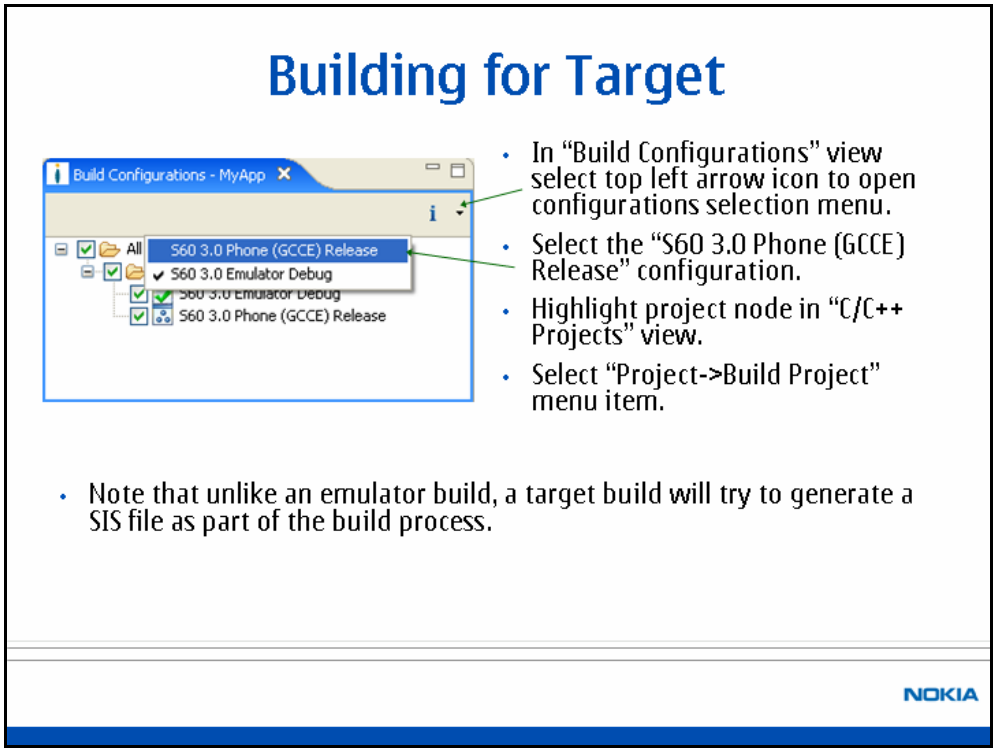
Under your application's thread node there are a number of child nodes which form the call stack for the thread. Each child node represents a stack frame in the call stack.

The Debug view also has a debugging toolbar that provides a number of useful functions including:

- Resuming thread execution.
- Stepping through the code.
- Terminating the debug session.

Note that when you finally terminate your debug session, the Carbide IDE will not automatically switch back to the Symbian perspective. You have to do this task manually.

Building for Target

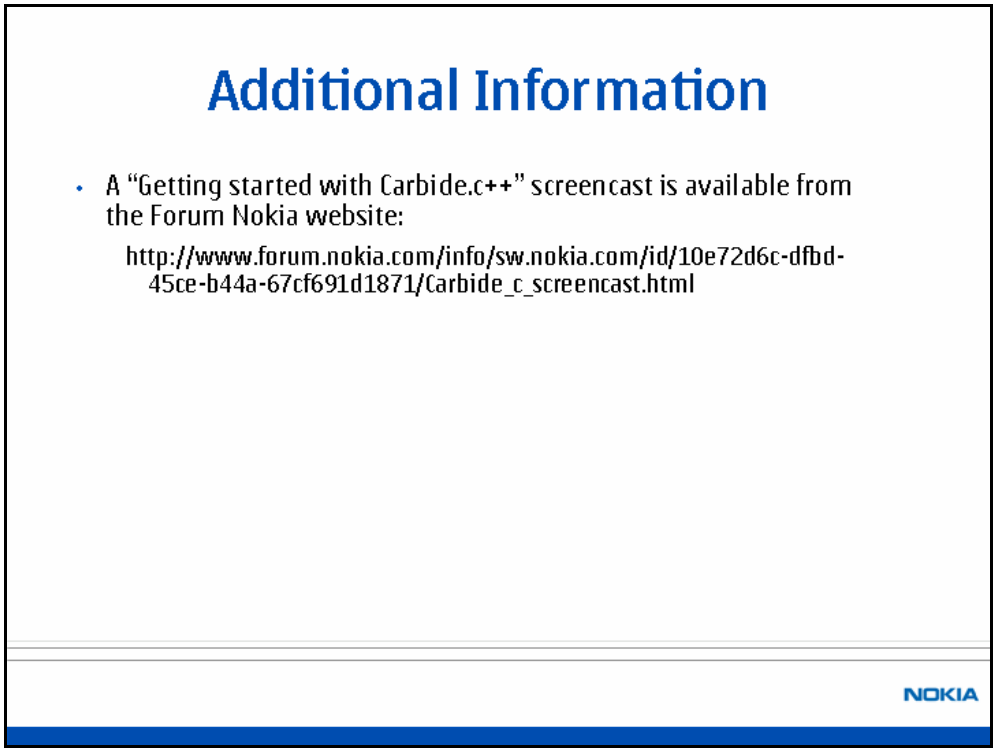


The final test for your application is to run it on a real device. For example, for an S60 3rd Edition device you must build your application using the "S60 3.0 Phone (GCCE) Release" configuration. To change to this configuration, first select the top-left arrow icon in the "Build Configurations" view. This will open the configurations selection menu. Select the "S60 3.0 Phone (GCCE) Release" configuration menu item. The configuration will now be changed.

You must now build your project for the new configuration. This is done in exactly the same way as building for the emulator configuration. Highlight the project node in "C/C++ Projects" view and select the "Project->Build Project" menu item.

Note that a "Phone" configuration also contains a build step for generating an installation (.SIS) file for your application. This SIS file is used to install your application onto a device. Simply upload the SIS file onto your device using Bluetooth, for example, and select the file to get the software installer to install it.

Additional Information



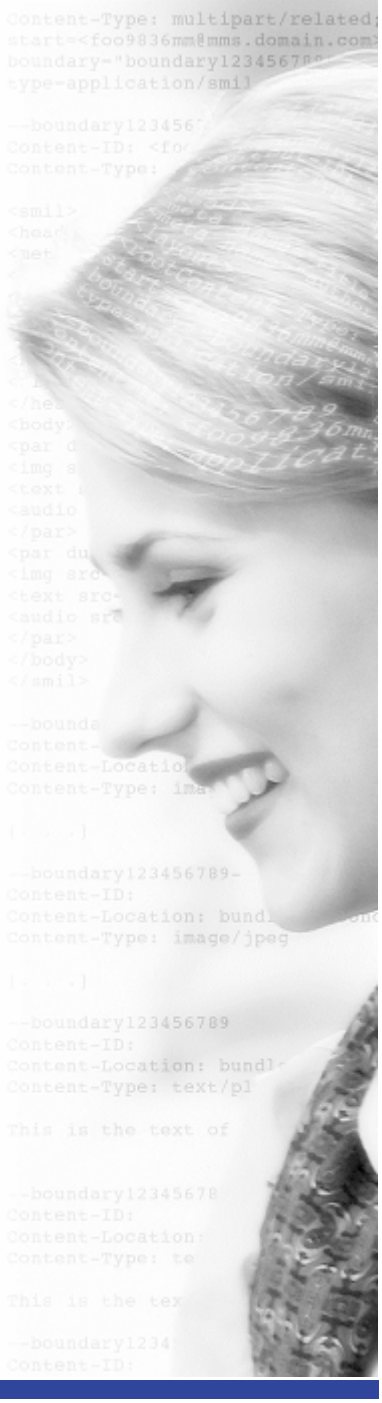
A above slides shows where to find additional information about Carbide.c++.

Developer
Training

Module #04303

Symbian OS Basics

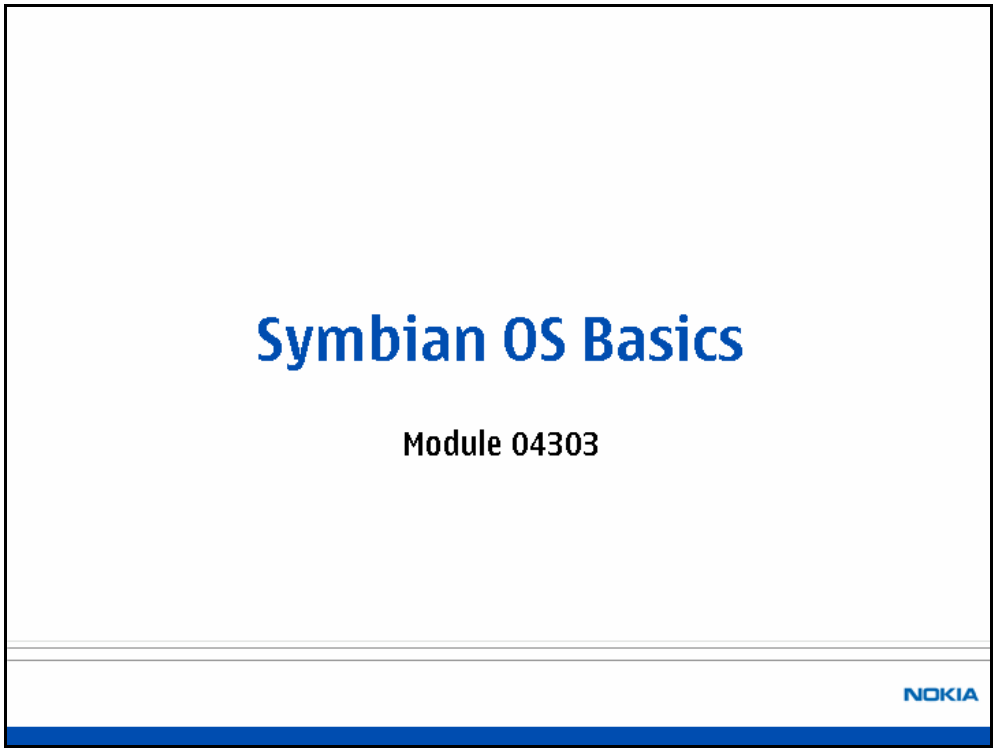
Contents



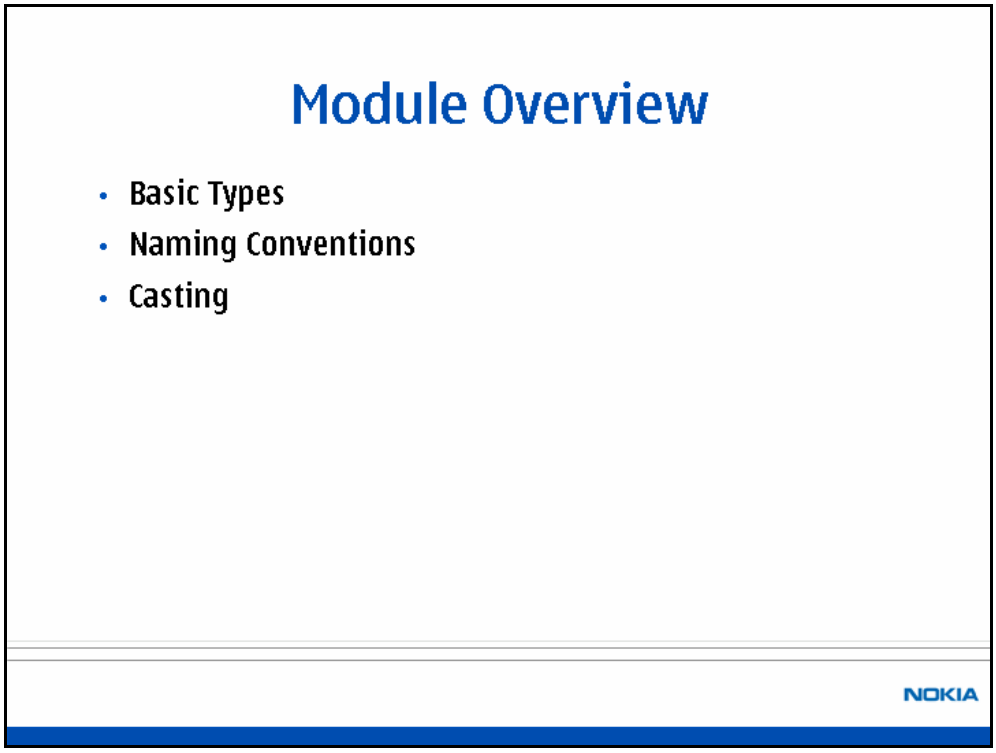
NOKIA

BO-EN-04300-20080108

Symbian OS Basics



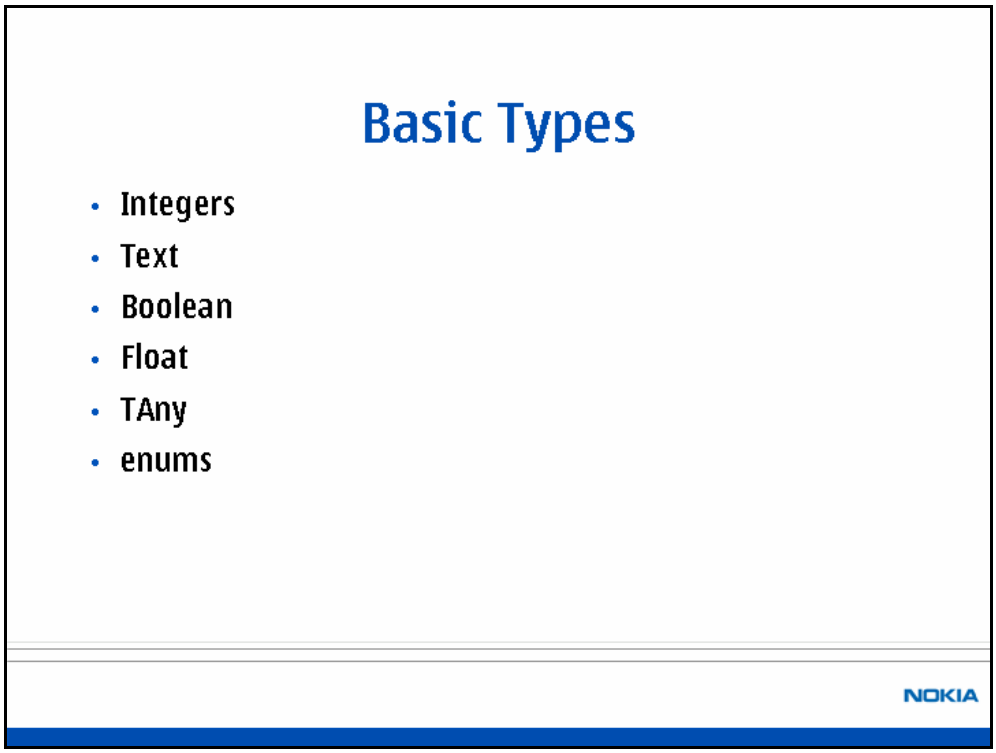
Module Overview



The topics covered in this module are:

- Basic types - these are redefined in Symbian OS.
- Naming conventions - classes, variables, functions and enumerations.
- Casting - Run time type information - not supported.

Basic Types

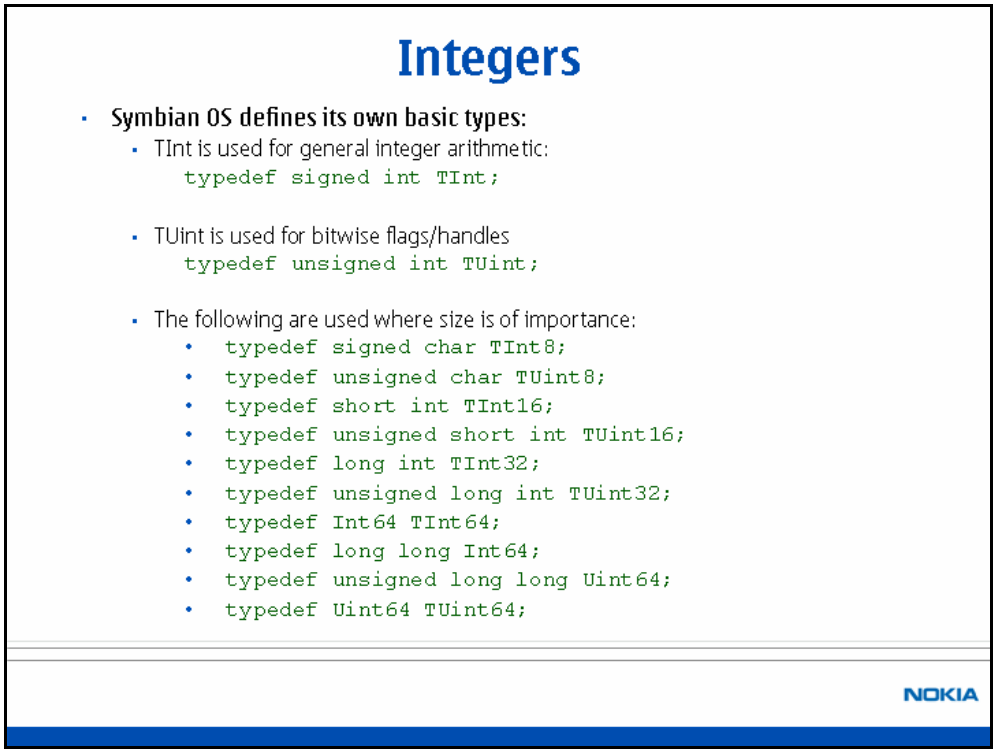


In Symbian OS programming, many of the basic C++ types are redefined. Although it is not compulsory to use them (in the sense that your project will still compile if you explicitly use the basic C++ types instead) they are almost always used in practice.

The main reasons adopting the new types are that:

- All the Symbian OS APIs use the redefined types instead of the basic C++ types, so using the basic C++ types would be inconsistent and confusing.
- Using them provides future proofing against future OS changes, such as Symbian OS moving from a 32-bit architecture to a 64-bit architecture.
- It is recommended practice by Symbian and is part of the “Symbian OS C++ Coding Standards”.

Integers



TInt

This is mapped to a C++ signed integer. It has a 32-bit representation at present. The definition is shown on the slide. The following code illustrates its use:

```
TInt arrayOfSquares[10];

for (TInt count = 0; count < 10; count++)
    {
    arrayOfSquares[count] = count * count;
    }
```

TUint

This is an unsigned integer of the natural machine word size (currently 32-bit). It should be used for counters and flags, which are manipulated using bit-wise and equality-comparison operations rather than arithmetic.

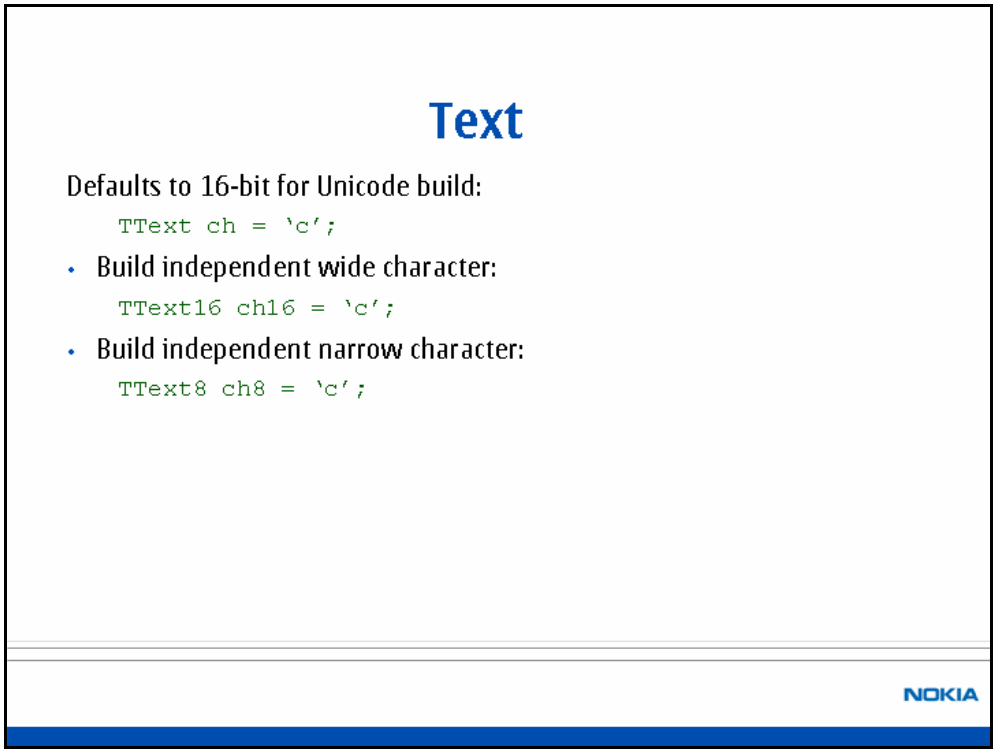
The following code sample shows the usage of some of the integer types:

```
const TUint oddMask = 0x00000001;
TInt sumOfOddSquares = 0;

for (TInt index = 1; index < 10; index++)
    {
    // bitwise AND determines whether index is odd number
    if (oddMask == (index & oddMask))
        {
        TInt square = index * index;
        sumOfOddSquares += square;
        }
    }
```

Integer Types Where Size Is Important

- `TInt64` defines a 64-bit signed integer type.
- `TUint64` defines a 64-bit unsigned integer type.
- `TInt32` defines a 32-bit signed integer type.
- `TUint32` defines a 32-bit unsigned integer type.
- `TInt16` defines a 16-bit signed integer type.
- `TUint16` defines a 16-bit unsigned integer type.
- `TInt8` defines an 8-bit signed integer type.
- `TUint8` defines an 8-bit unsigned integer type.

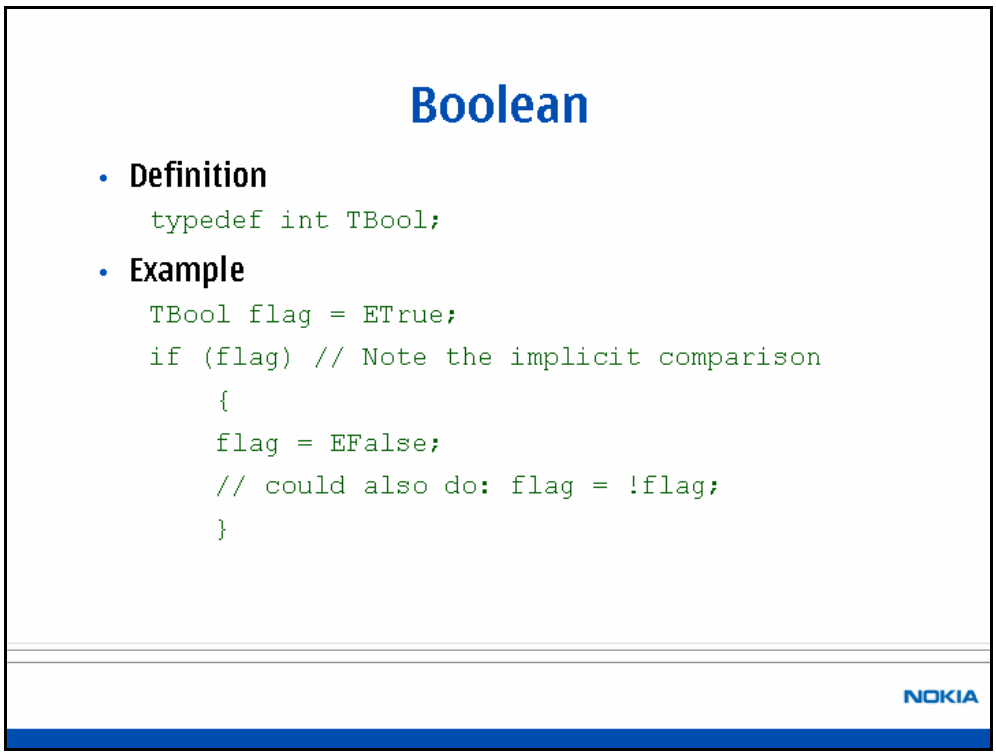
Text



The text types are not commonly used in Symbian OS programming. Instead, descriptors are used for string handling (descriptors are classes that encapsulate the string data).

When the `TText` types are used, it is usually for single characters. The slide illustrates the usage of the `TText`, `TText16` and `TText8` types. Note that for most purposes the type that does not specify the width is used as it defaults to a 16-bit type for a Unicode build (the standard build).

Boolean



The `TBool` type should be used for boolean variables. For historical reasons it is implemented as a `TInt`.

Two enumerated values, `ETrue (=1)` and `EFalse(=0),` are also defined. One of these should always be used to set the value of a `TBool` variable. Since C++ interprets any non-zero value as true, you should refrain from comparing a `TBool` variable with `ETrue`. To get round this comparison problem with `ETrue` implicit comparisons are used.

For example you should use:

```
if (iBool)
    {
    …
```

instead of:

```
if (iBool==ETrue)
    {
    …
```

Comparisons using `EFalse` are fine but implicit comparisons are usually preferred; i.e. use:

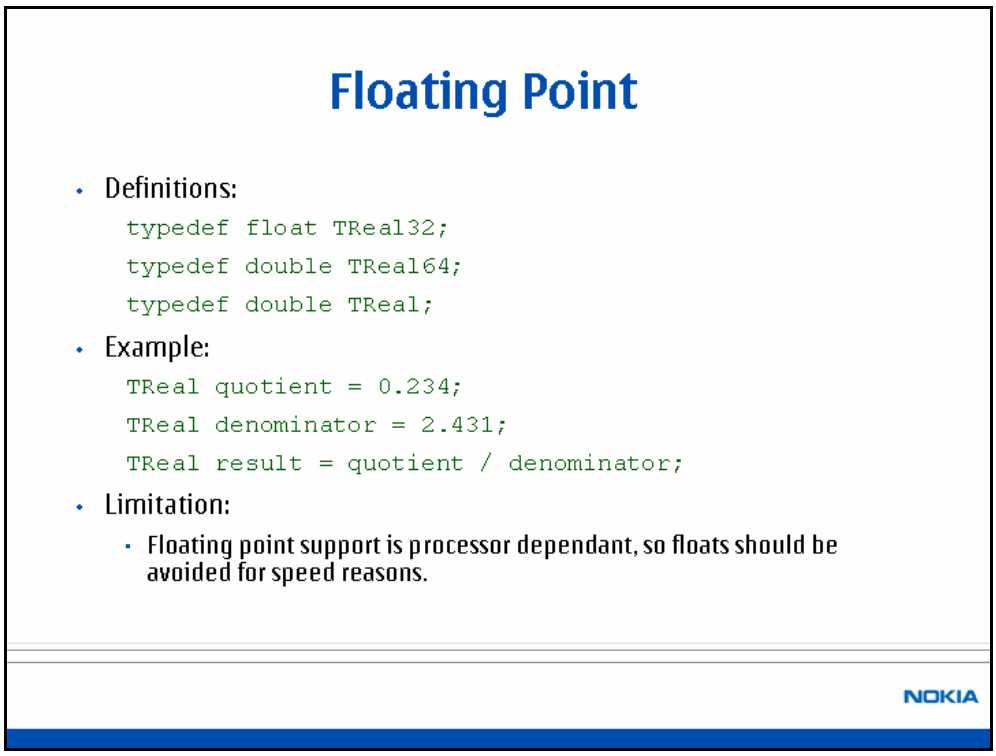
```
if ( !iBool )
    {
    …
```

instead of:

```
if (iBool==EFalse)
```

```
{
...
```

Floating Point



TReal, TReal64

These are double-precision floating point. This floating-point is the IEEE754 64-bit representation and should be used in most cases. Floating-point support is processor dependent, so operations should be done using integer arithmetic if possible (for instance, most GUI calculations). Floating-points should be used only when the problem demands it (for instance, a spreadsheet application).

TReal32

This is a 32-bit floating point. This is smaller and quicker, but should only be used when space and/or time are at a true premium, since its precision is unsatisfactory for many applications.

General Rule

Floating point maths may be slow on mobile devices because they do not always contain a Floating-Point Unit (FPU). In other words, floating-point operations may have to be done internally using integer arithmetic, which is very inefficient. For this reason it is often worth avoiding using a floating-point number altogether and using an alternative integer algorithm.

For example, the following code using `TReals`:

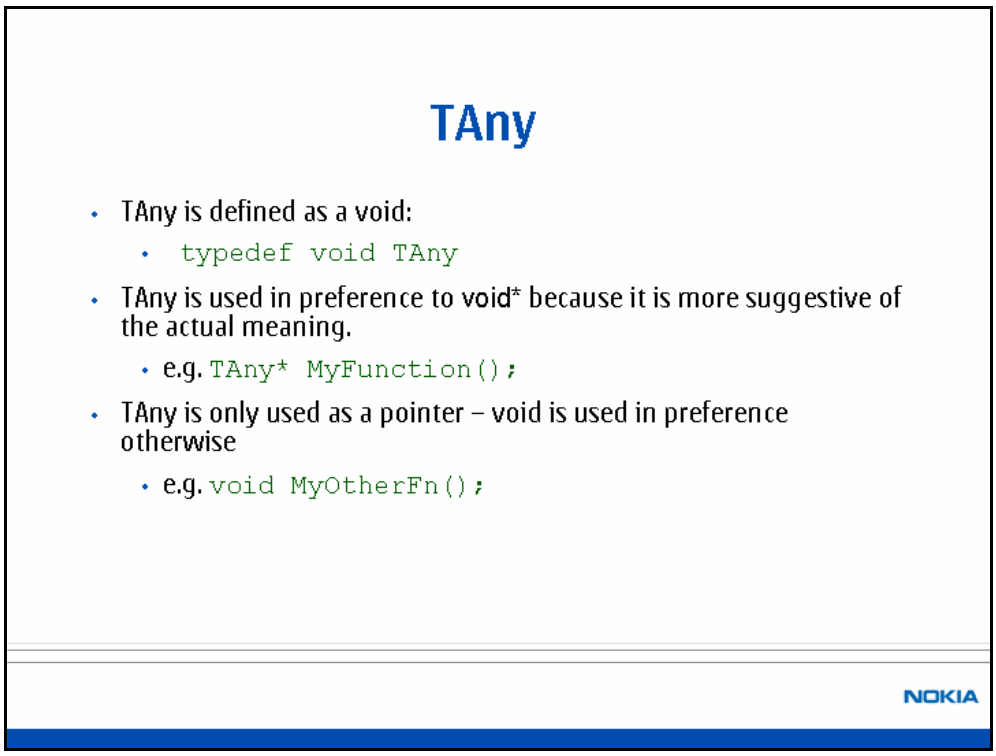
```
TInt SomeFunction(TInt aTop, Tint aBottom)
    {
    TReal a = (TReal)aTop;
    TReal b = (TReal)aBottom;
    TReal c = a/b+0.5;
    TReal result;
```

```
    Math::Round(result,c,0);

    return (TInt)result;
    }
```

could be replaced by the following code (using only integers) instead:

```
TInt SomeFunction(TInt aTop, Tint aBottom)
    {
    return((2*aTop+aBottom)/(2*aBottom));
    }
```

TAny



`TAny*` is used in many APIs throughout Symbian OS, for example in memory manipulation APIs (which will be covered later). For example, the static memory function, `Mem::Copy()` is specified as:

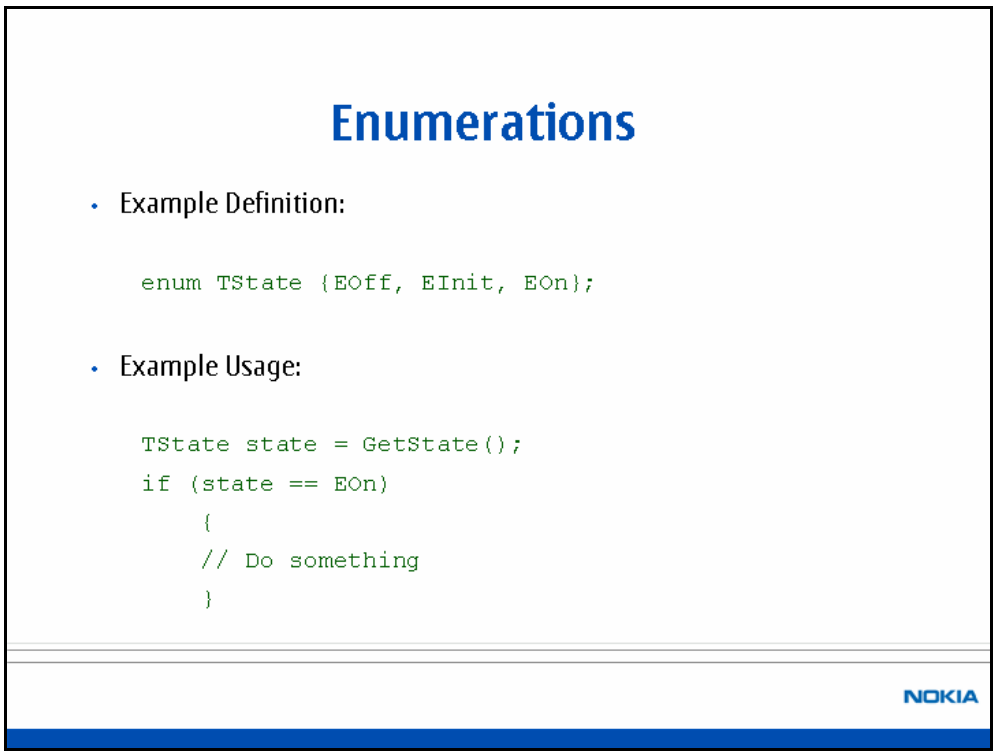
```
static TUint8* Copy(TAny* aTrg,const TAny* aSrc,TInt aLength);
```

It is used in the following code sample:

```
const TUint8 binaryData[] =
    {0xa1, 0xf2, 0x8c, 0xd8, 0x0f, 0xf0};
TUint8 binaryDataCopy[sizeof(binaryData)];
Mem::Copy(binaryDataCopy, binaryData, sizeof(binaryData));
```

Note that `TUint8*` is converted implicitly to `TAny*` without the need for a type cast.

Enumerations

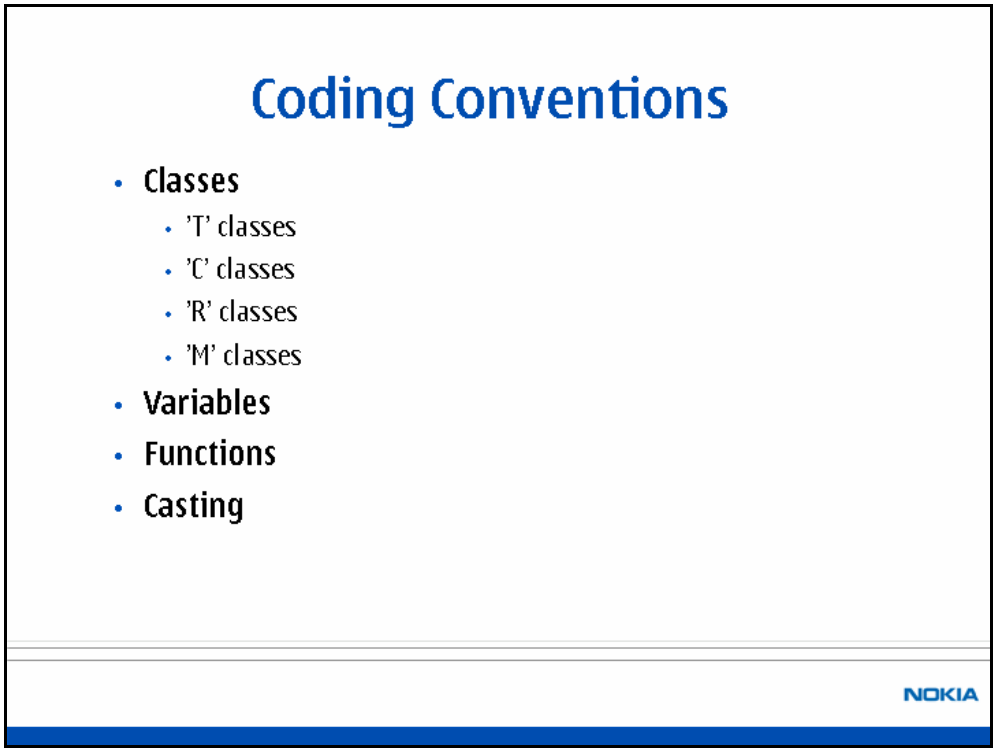


Enumeration types are and members should always have meaningful unambiguous names. They are frequently defined within class scope. For example:

```
class TSomeClass
    {
    public:
        enum TState {EOn, EInit, EOff};
        …
    };
```

Note that enumeration types should always begin with a capital ‘T’ and enumerated values should always begin with a capital ‘E’.

Coding Conventions



The next section looks at the coding conventions used by Symbian OS programmers. These differ considerably from other naming conventions, for example Microsoft VC++/MFC, Hungarian notation.

In Hungarian notation all automatic variable names are prefixed with an indication of the type (such as iNumber for an integer and bFlag for a Boolean). In Symbian programming Hungarian notation is not used and the name of the automatic variable should just be the description.

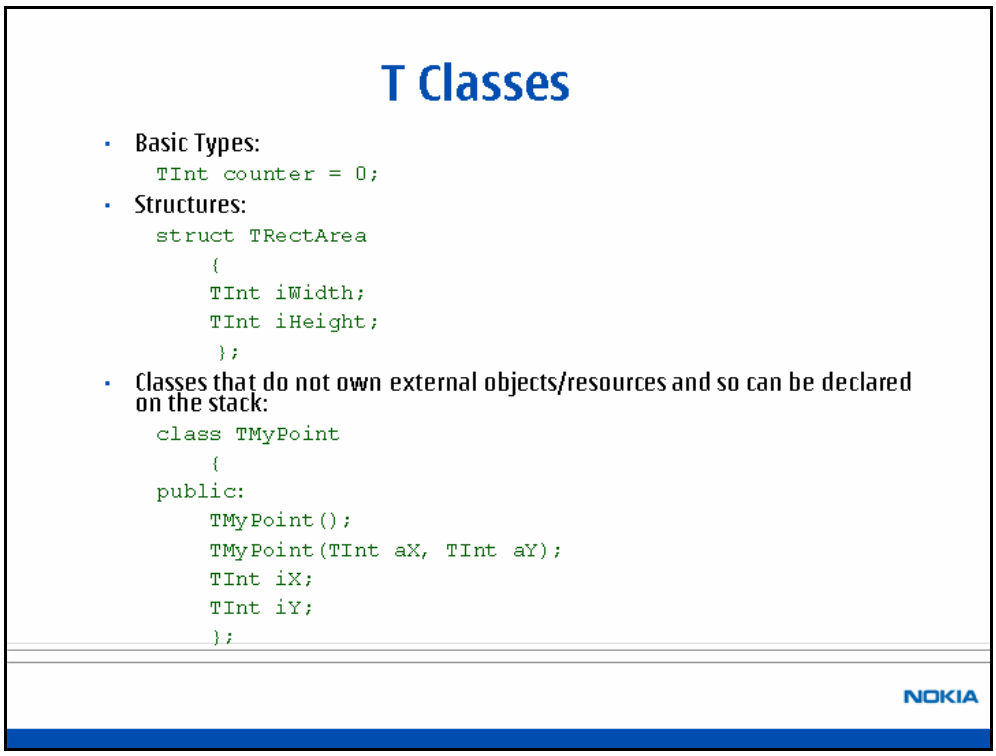
The topics covered in this section are as follows:

The class naming conventions will be examined. This all depends on the type of class in use, how the class is allocated, and so on. There are four types of classes:

- Those that can be declared on the stack.
- Those that must be declared dynamically on the heap.
- Those that access shared resources (for example files).
- Those that have no implementation and define an interface

Additionally, the naming conventions of variables and functions and the use of casting operators are examined.

T Classes



The most fundamental types are value types. These are given type, or class, names beginning with 'T'. 'T' types contain their value. They do not own any external object, either directly (by pointer) or indirectly (by handle). Therefore, they do not need a destructor to cleanup resources. 'T' types may be allocated either on the stack (that is locally as C++ automatic variables) or as members of other classes. However, note that the default stack size is 8KB in Symbian OS, so large 'T' classes should be used very carefully as automatic variables. Structure types mostly begin with 'T'. An example of a 'T' class and its use is as follows:

```
class TVersion
    {
    public:
        IMPORT_C TVersion();
        IMPORT_C TVersion(TInt aMajor,TInt aMinor,TInt aBuild);
        IMPORT_C TVersionName Name();
    public:
        TInt8 iMajor;
        TInt8 iMinor;
        TInt16 iBuild;
    };
```

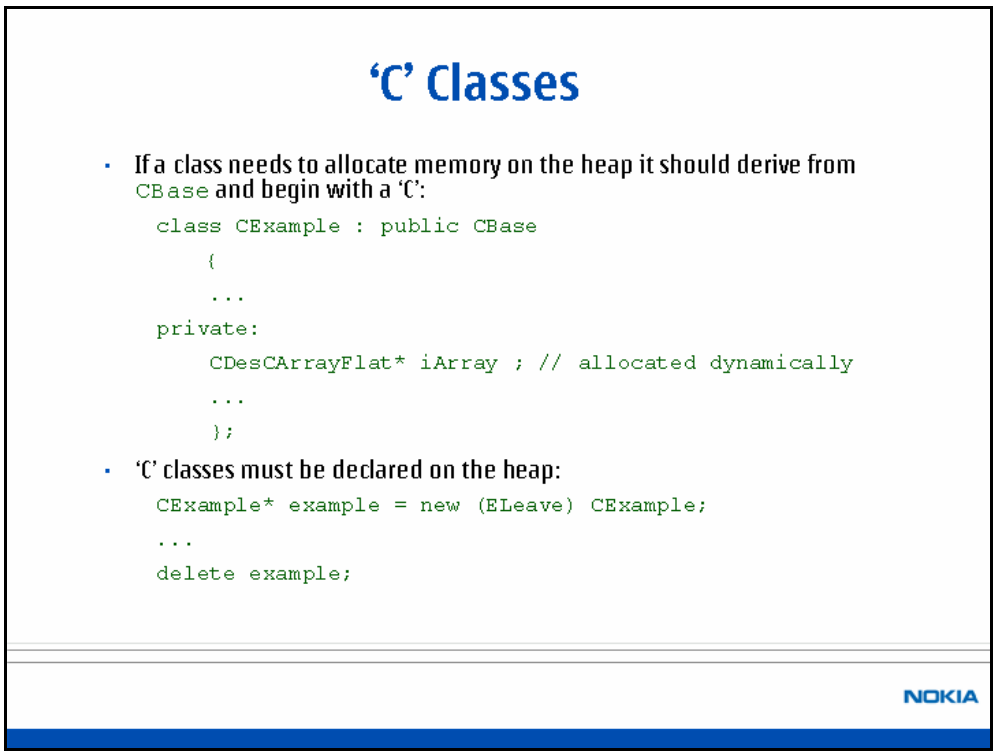
Note that the IMPORT_C macro specifies that the user of that API is importing the function from a DLL.

The class could be used as follows:

```
    // User::Version() returns the version of the OS
    // osVersion is an automatic variable
    TVersion osVersion = User::Version();
    if (1 == osVersion.iMajor)
        {
```

```
    // Do something
    }
// No need to worry about de-allocating TVersion
// because it is on the stack
```

‘C’ Classes

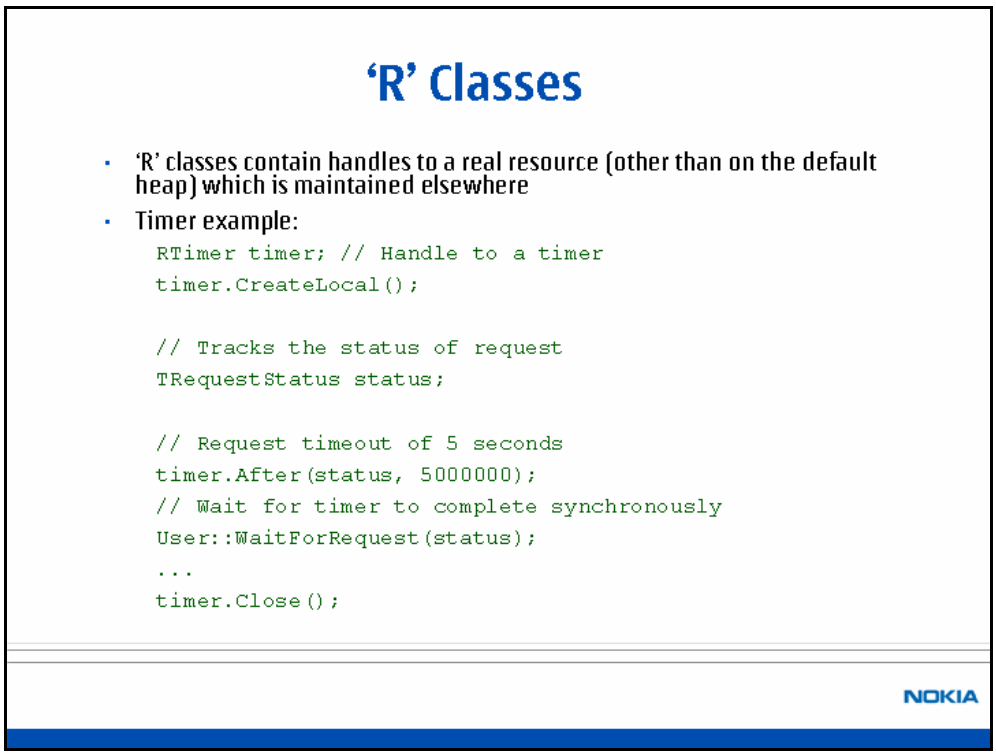


If a class needs to dynamically allocate any memory or own another class, it should derive, directly or indirectly, from `CBase` (a built-in Symbian OS class), and its class name should begin with ‘C’. `CBase`-derived classes have the following properties:

- They are allocated on the heap (dynamically allocated) — not on the stack, and not as members of other classes.
- The allocator used for this class hierarchy initializes all member data to binary zeroes.
- They are passed by pointer, or reference, and so do not need an explicit copy constructor or assignment operator unless there is clear intention that a particular class supports copying.
- They have a virtual destructor, which is used for standard cleanup processing.
- They support two-phased construction – this will be covered later in the section.

An example of the use of ‘C’ classes is provided later on, since it includes associated concepts such as two-phase construction and leaves which have not been covered yet.

‘R’ Classes

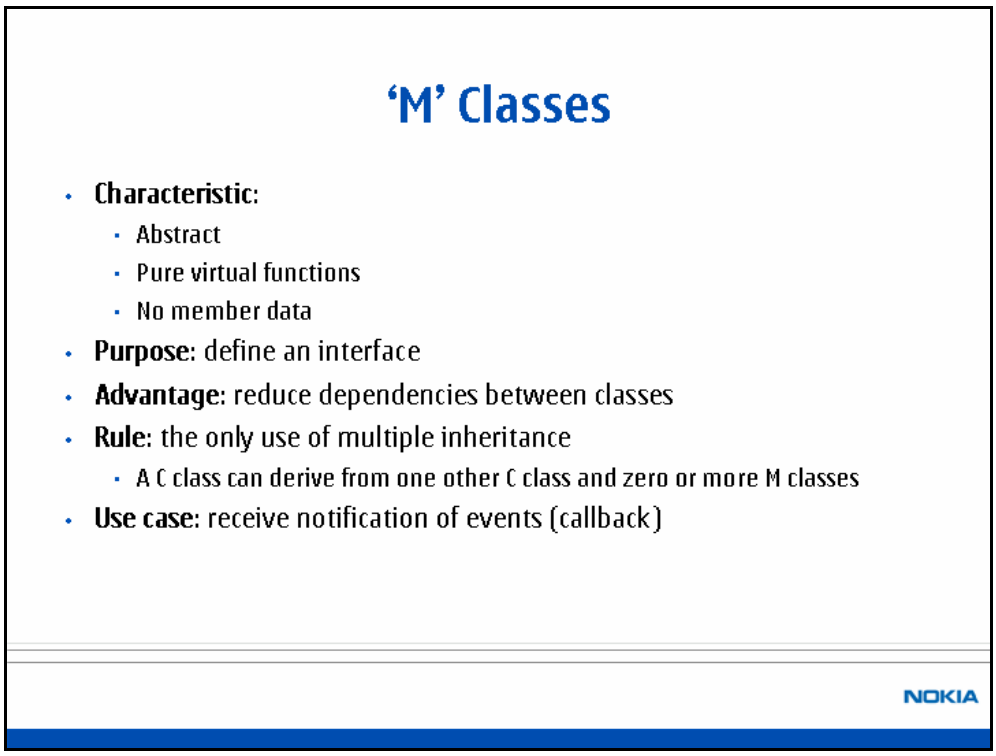


These classes are used to access system resources, for example files, via handles. The following are characteristics of ‘R’ classes:

- They contain a handle that is used to pass on requests to other objects.
- They are opened using an "`Open()`" function particular to the ‘R’ class, and closed using a "`Close()`" function particular to the class. An ‘R’ object must be closed once if it has been opened.
- They may be freely bit-wise copied.
- They have no explicit constructor, destructor, copy constructor or assignment operator.

The example on the slide illustrates sample code for a timer. Note: within a GUI application `User::WaitForRequest()` would not be used for such a period since it would mean that the user would not be able to select, for example, menu items, for 5 seconds. Instead active objects (covered in the module entitled “The Active Object Framework”) would be used to notify the application when the timer event occurred.

‘M’ Classes



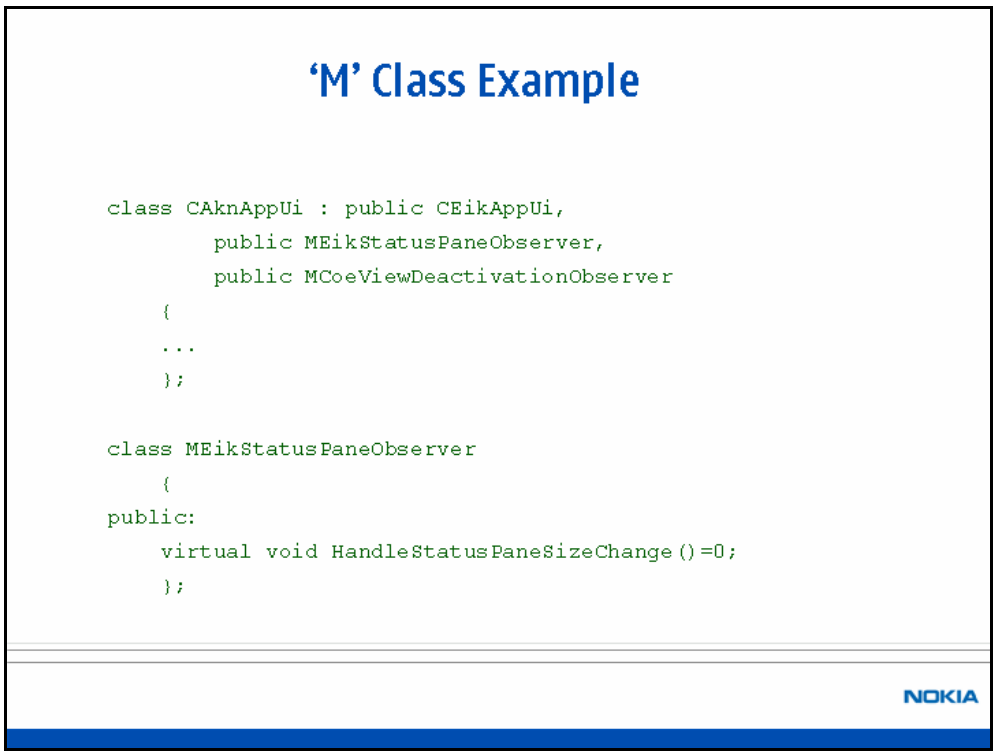
‘M’ classes have the following restrictions:

- They should contain no member data.
- They should not contain constructors or destructors, or overloaded operators.

‘M’ classes usually contain pure virtual functions that define a fully abstract interface. These act exactly as interfaces in Java. Some ‘M’ classes implement some or all member functions, though within the restrictions given above. They tend to be very useful classes, especially for event notifications and to reduce dependencies between classes.

‘M’ classes provide the only use of multiple inheritance in Symbian C++ programming. The old technical term for them was “mixin”, hence the use of the letter ‘M’. However they are more commonly referred to as “interfaces” now.

‘M’ Class Example



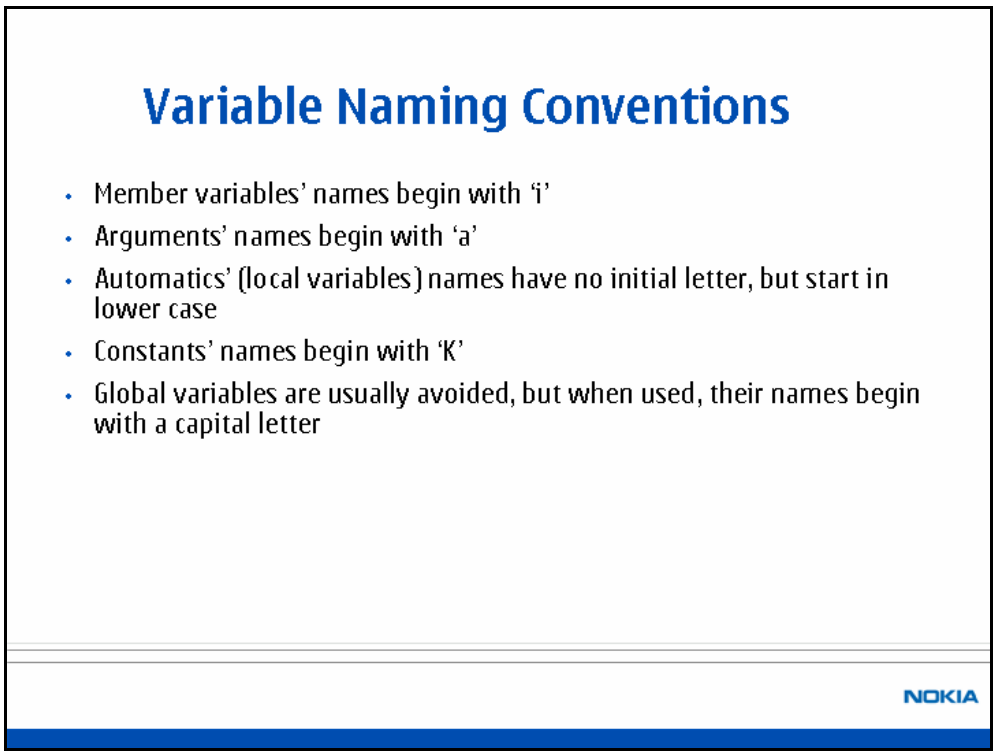
The slide shows a partial class definition for the standard S60 application UI class, `AknAppUi`. (The topic of application UI classes is covered in a later module). This class implements the `MEikStatusPaneObserver` interface so that it can be notified of changes in size to the status pane (the top portion of the screen containing the application title).

Any ‘C’ class that derives from `MEikStatusPaneObserver` must override the `MEikStatusPaneObserver::HandleStatusPaneSizeChange()` function because it is pure virtual. This provides a mechanism for the derived class to receive notification of an event. Additionally the class that sends the event to the derived class only needs to know about the interface and not the whole of the class, thereby reducing compilation dependencies.

Below is the partial definition for the base class for the status pane, `CEikStatusPaneBase`. This has a `MEikStatusPaneObserver*` member variable so it can notify an observer class (in this case the application UI class on the slide) when it resizes. There is also a `SetObserver` function so that classes can register to receive the notification.

```
class CEikStatusPaneBase : public CBase
     {
    public:
    ...
        inline void SetObserver(MEikStatusPaneObserver* aObserver);
    ...
    private:
    ...
        MEikStatusPaneObserver* iObserver;
    ...
    };
```

Variable Naming Conventions



The naming convention for member variables is very important for cleanup during destruction. All member variable names are prefixed with a lower case 'i'. It makes member variables easier to identify and reduces the risk of them not being cleaned up properly.

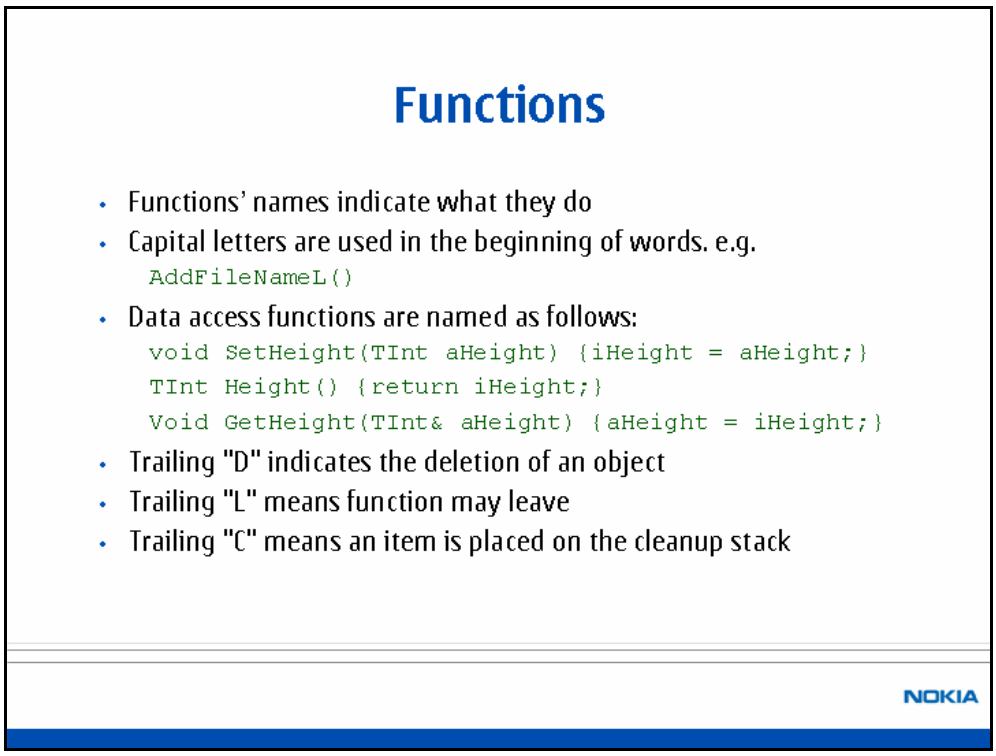
Global and static member variables are not allowed in applications and shared libraries. If they are used, there will be an error in the target build. However, it is possible to have global and static member constants – the system wide constants are obviously global.

Example code showing the use of variables is shown below. A class is used to compare values passed to it and return –1, 0, or 1 depending on whether the comparison value is less than, equal to or more than the value held in the class.

```
class TMyValue
    {
    public:
        TMyValue() {iValue = 0;} // defined inline
        TMyValue(TInt aValue){iValue = aValue;}
        void Set(TInt aValue){iValue = aValue;}
        TInt Compare(TInt aCompareValue);
    private:
        TInt iValue;
    };

TInt TMyValue::Compare(TInt aCompareValue)
    {
    TInt KMaxVal = 100; // constant
    TInt retVal = 0; // automatic variable
    if (iValue > KMaxVal)
        {
        retVal = -2;
```

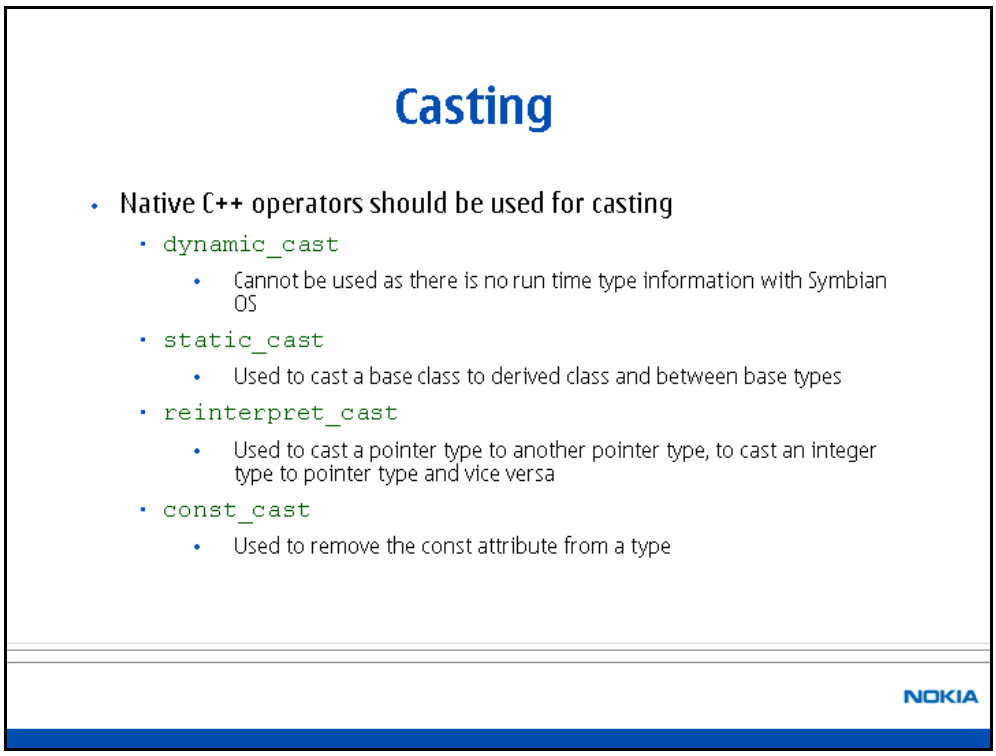
```
    }
else if (iValue < aCompareValue)
    {
    retVal = 1;
    }
else if (iValue > aCompareValue)
    {
    retVal = -1;
    }
return retVal;
}
```

Functions



The slide illustrates the style of function names. Note the access function naming convention for 'Get' functions. If the value is just returned, 'Get' is omitted from the name. If the value is passed back by a reference parameter, 'Get' precedes the function name. The final three points on the slide refer to material that has not been covered yet. Leaves and the cleanup stack will be covered in the module entitled "Memory and Resource Management".

Casting



Casting is used to convert between classes and types. C casts are still legal in Symbian OS, but C++ casts should be used in preference since they remove any ambiguity that may be caused by using the old style casts.

A `static_cast` example:

```
TInt intValue = 0xff;
// There would be a compiler warning if the cast was not used:
TUint8 byteValue = static_cast<TUint8>(intValue);
```

A `const_cast` example is shown below. A dummy class `TMyIntPtr` requires a `TInt*` for construction:

```
class TMyIntPtr
    {
    public:
        TMyIntPtr(TInt* aPtr){iPtr = aPtr;}
    private:
        TInt* iPtr;
    };
```

However, construction with a `const TInt*` gives a compilation error, so a `const_cast` is needed to remove the const:

```
    const TInt myVal = 1;
    TMyIntPtr val(const_cast<TInt*>(&myVal));
```

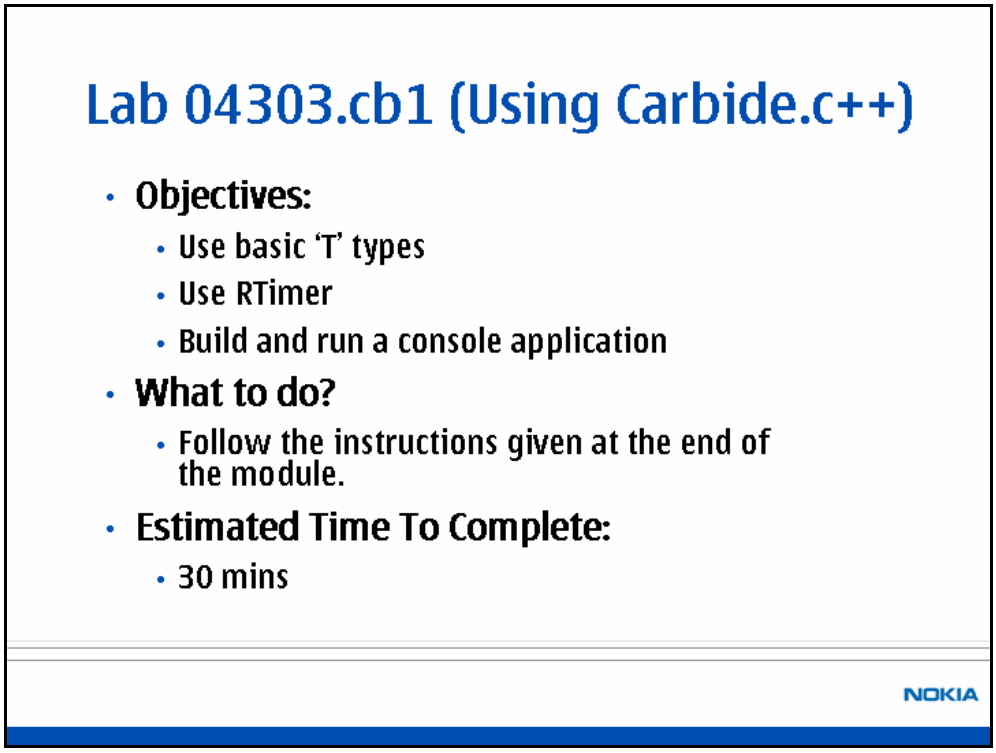
This is a potentially dangerous thing to do because if `TMyIntPtr` attempted to modify the contents of the pointer (that is `myVal`), a panic would occur (a panic kills an

application as soon as possible after an error is detected, and returns an error code which is of some use to the developer).

A `reinterpret_cast` example:

```
TUint32 fourBytes = 0;
TUint8* bytePtr = reinterpret_cast<TUint8*>(&fourBytes);
bytePtr++; // move to the second byte of fourBytes
*bytePtr = 0xff; // fourBytes now == 0x0000ff00;
```

Lab 04303.cb1 (Using Carbide.c++)



Lab 04303.cb1

Overview

Title:	Using Symbian OS Basic Types and Adding RTimer Functionality		
Overview:	This lab demonstrates the use of some basic Symbian types in a console application and shows how to implement a basic timer.		
Objectives:	To understand how to: • Use basic Symbian types. • Understand what is required to implement a basic timer. • Build and run a console application.		
Compatible IDE(s):	Carbide.c++ v1.2		
Compatible SDK(s):	S60 3rd Edition, S60 3rd Edition (MR), S60 3rd Edition FP1		
App. Type:	Console	**App. Name:**	SymbianOSBasicsLab.exe
Starter Code Provided:	Yes	**Solution Code Provided:**	Yes
Estimated Time To Complete:	30 minutes		

Lab Instructions

Exercise 1 – Preliminary Steps

Task 1.1 – Checks

1. Ensure that a compatible SDK has been installed. A list of compatible SDKs to use with this lab is given in the table in the Introduction.
2. Ensure that a compatible IDE has been installed. A list of compatible IDEs to use with this lab is given in the table in the Introduction.
3. Ensure that the lab files are located in the folder `C:\Labs\Lab_04303.cb1\`. If this is not the case please refer to the setup guide for details of how to obtain them.

Task 1.2 – Make sure the emulator is in text shell mode

1. Open Windows Explorer.
2. Navigate to the file named epoc.ini, located in the `<EPOCROOT>\epoc32\data` folder. Here, `<EPOCROOT>` is a placeholder for the root folder location of your SDK.
3. Open the epoc.ini file using Notepad.
4. Check that the file contains the "textshell" statement. If the statement is NOT present in the file, then add it to the beginning of the file as shown by the text in bold below:

```
textshell
<Other statements in epoc.ini file>
...
```

5. Press 'Ctrl' + 'S' to save the contents of the file (if you altered it) and close Notepad.
6. Close Windows Explorer.

Task 1.3 – Carbide Setup

1. Launch the Carbide.c++ IDE. Depending on your Carbide.c++ settings the IDE will either display the "Workspace Launcher" dialog or open the last used workspace. If the latter occurs open the "Workspace Launcher" dialog by selecting the "File -> Switch Workspace..." menu item.
2. In the workspace launcher window, create a new workspace by typing `C:\Labs\<Workspace>` in the Workspace field. Here `<Workspace>` is a placeholder for a folder name of you choosing that does not already exist and does not contain spaces.
3. Click the OK button.
4. Close the Welcome window after it appears when the workspace is opened.

Task 1.4 – Importing the project

1. In the Carbide.c++ IDE, select the "File > Import" menu item. The Import dialog appears.
2. Select "Symbian OS bld.inf file" from the "Symbian OS" subfolder as the import source and click the Next button. A new screen appears on the Import dialog.
3. Enter `C:\Labs\Lab_04303.cb1\starter\group\bld.inf` in the "bld.inf file" field. Alternatively, click the "Browse..." button and then navigate to and select the correct file via the dialog. When this is done a list of configurations appears in the "SDKs and Build Configurations pane".
4. Expand the list of configurations for your installed SDK and tick the checkbox next to the "Emulator Debug" configuration. (Note that other configurations may be chosen in addition to this one, but they are not used in this lab.) Then select the "Next" button.
5. Ensure the MMP file and all the make files are selected on the "MMP Selection" screen. Then select the Next button. The Project Properties screen appears.
6. Change the name in the Project field to "SymbianOSBasicsLab" (without the quotes) and click the Finish button to complete the import.
7. The Import dialog closes and a new project called "SymbianOSBasicsLab" appears in the "C/C++ Projects" view.

Exercise 2 – Using some basic Symbian OS types

Task 2.1 – Editing the application

1. In the Carbide.c++ IDE, double click on the source file node, `\SymbianOSBasicsLab\SymbianOSBasicsLab.cpp`, in the "C/C++ Projects" view. A text editor window is opened showing the contents of the file.
2. In the `UseBasicTypesL()` function, under the "Edit 1" comment, declare a variable called `sum` of type `TInt` that adds `KOne` and `KTwo` together.
3. Uncomment the line below the "Edit 2" comment.
4. Under the "Edit 3" comment, declare a variable called `flag` of type `TBool` and initialise it to `EFalse`.
5. Uncomment the 5 lines below the "Edit 4" comment. These print out the value of the flag if it is `EFalse`.

6. Under the “Edit 5” comment, toggle the value of `flag`. Hint: use `flag = !flag;`
7. Uncomment the 5 lines below the “Edit 6” comment. These print out the value of the `flag` if it is `ETrue`.
8. Select “File -> Save” to save the changes to the source file.

Task 2.2 – Building and running the application

1. Build the application for the Emulator Debug configuration of the SDK you are using by right clicking on the SymbianOSBasicsLab project in “C/C++ Projects” view and select the “Build Project” or “Rebuild Project” menu item. The project should build successfully with no errors.
2. Run the console application by right clicking on the SymbianOSBasicsLab project in “C/C++ Projects” view, and selecting the “Run As -> Run...” menu item. The Run dialog appears.
3. Click the New button on the Run dialog. The Run dialog screen changes to allow the editing of settings on a new run configuration. Accept all the default settings apart from the “Emulator or host application” field which should be made empty; i.e. its contents should be removed.
4. Click the Run button on the Run dialog. This will run the console application. The application will execute until input from the user is required. Figure 1 shows a screen shot of the application at this point.
5. Press the space key to continue application execution. The comment mentions that a timer will be started but this has not been implemented yet, so the application executes until further input is required from the user. Figure 2 shows a screen shot of the application at this point.
6. Press any key. The application completes its execution and the emulator window closes.

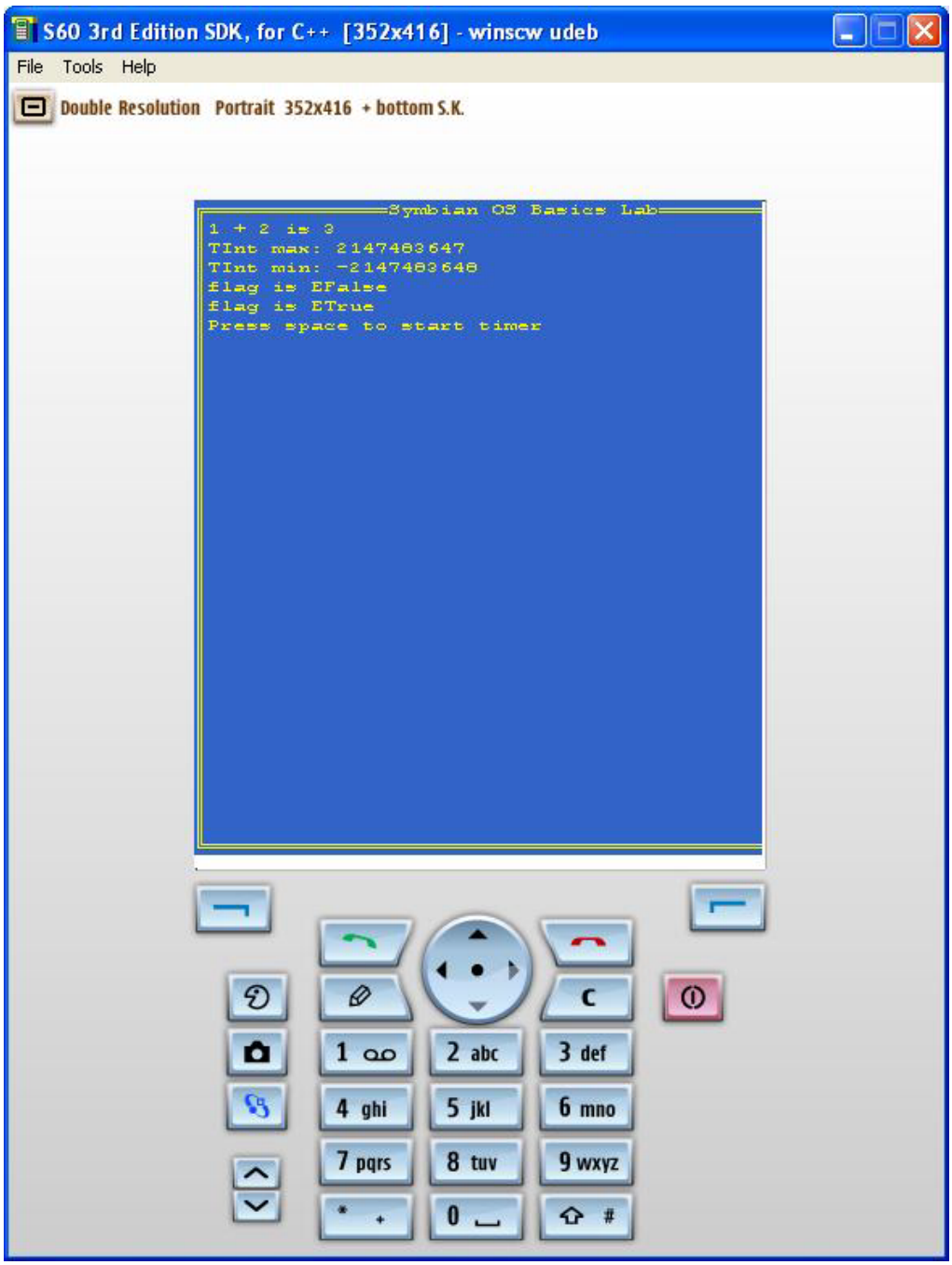


Figure 1 Emulator window after running the console application

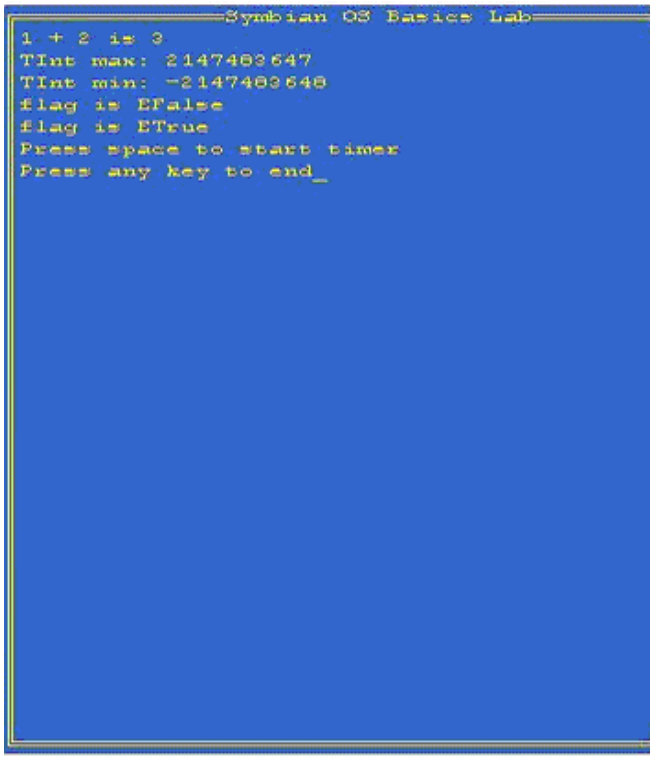


Figure 2 Emulator screen shot after pressing space key

Exercise 3 - Adding RTimer Functionality

Task 3.1 – Editing the application

1. Select the "text editor" window showing the contents of the SymbianOSBasicsLab.cpp file.
2. Under the "Edit 7" comment declare a variable called `keyVal` of type `TText` and set it to the value of key.
3. Uncomment the line below the "Edit 8" comment. This prints out the key entered by the user (if it isn't the space key).
4. Uncomment the line below the "Edit 9" comment to request a timeout of `KTimerInterval` micro seconds.
5. Uncomment the line below the "Edit 10" comment to wait synchronously for the timer to complete. Note: This is generally bad practice in GUI applications as it prevents user key presses from being handled.
6. Uncomment the 2 lines below the "Edit 11" comment. These print the timeout value to the screen.
7. Select "File -> Save" to save the changes to the source file.

Task 3.2 – Building and running the application

1. Rebuild the application and launch the emulator as before. The application will execute until input from the user is required. At this stage, the output of the application is identical to the output of the previous run. See Figure 1 for a screen shot of the emulator.
2. Type any key apart from the space key. A string, indicating the key typed, is displayed in the emulator window. Type a few more keys and see that further strings, each indicating the particular key typed, are also displayed. Figure 3 shows a screen shot of the application after typing a number of different keys.

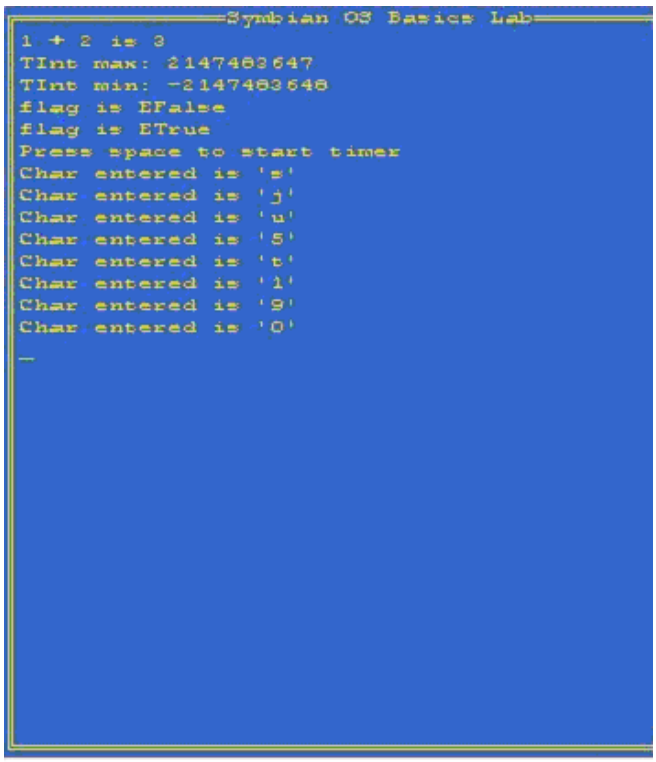


Figure 3 Screen shot of emulator window after typing a few keys

3. Type the space key. This begins a 5 second timer during which time the application thread is blocked. After this time execution continues and a message is displayed to the screen. Execution then halts as further user input is required. Figure 4 shows a screen shot of the application at this point.

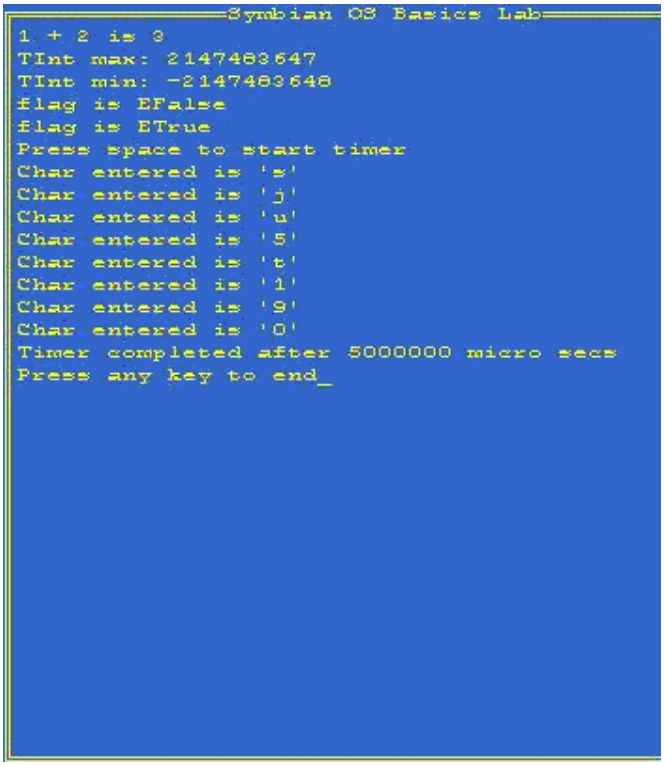


Figure 4 Screen shot of emulator window after timer has expired

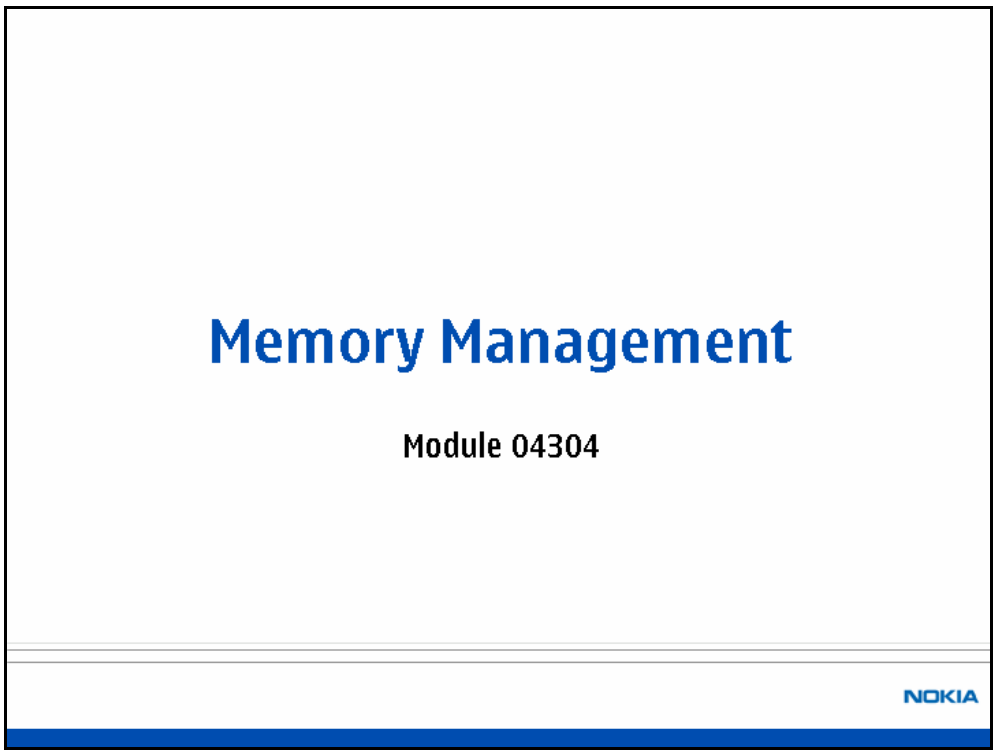
4. Press any key. The application completes its execution and the emulator window closes.
5. Close the Carbide.c++ IDE. This completes the lab.

Module #04304

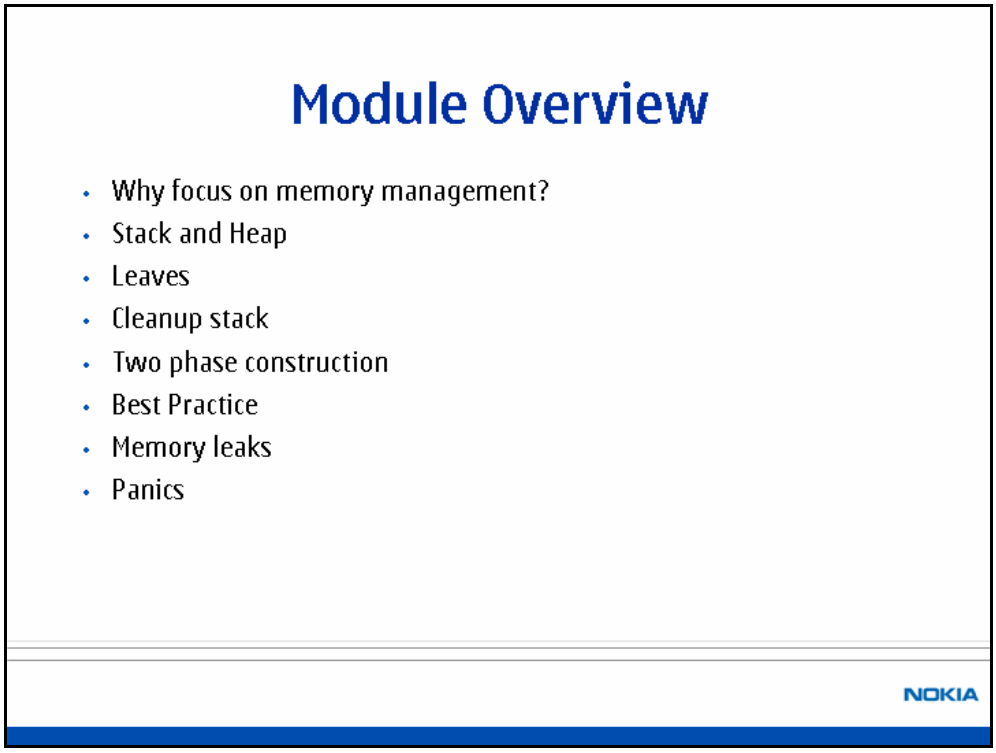
Memory Management

Contents

Memory Management

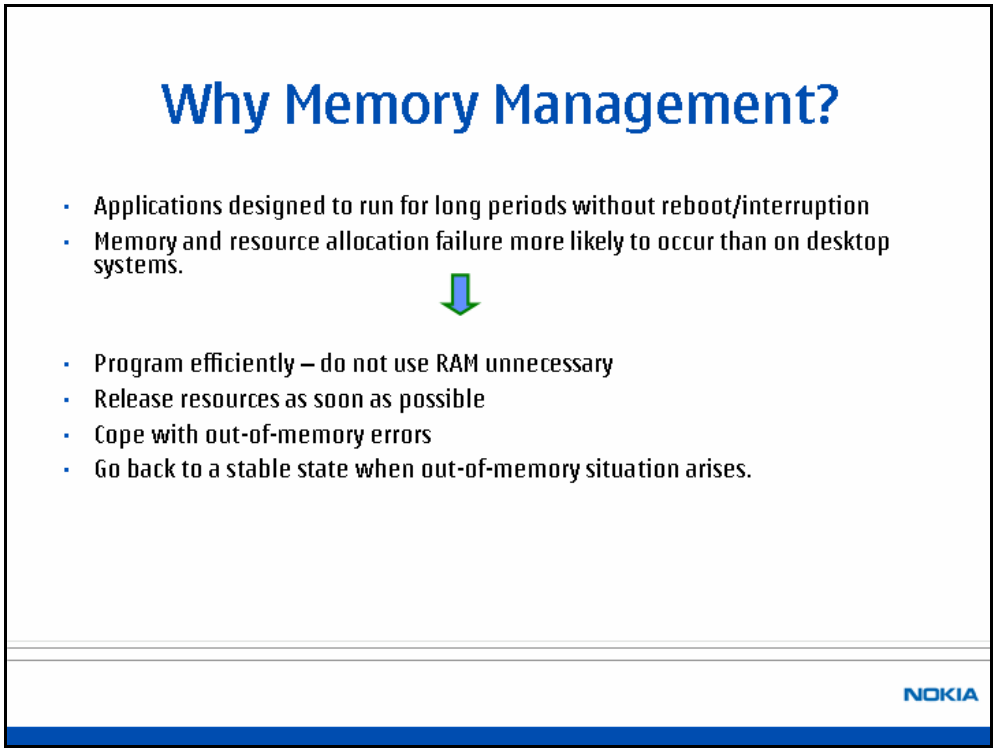


Module Overview



This slide presents the topics that will be covered in this module.

Why Memory Management?



An essential characteristic of Symbian OS is that it is designed for devices where the amount of memory and resources available is limited.

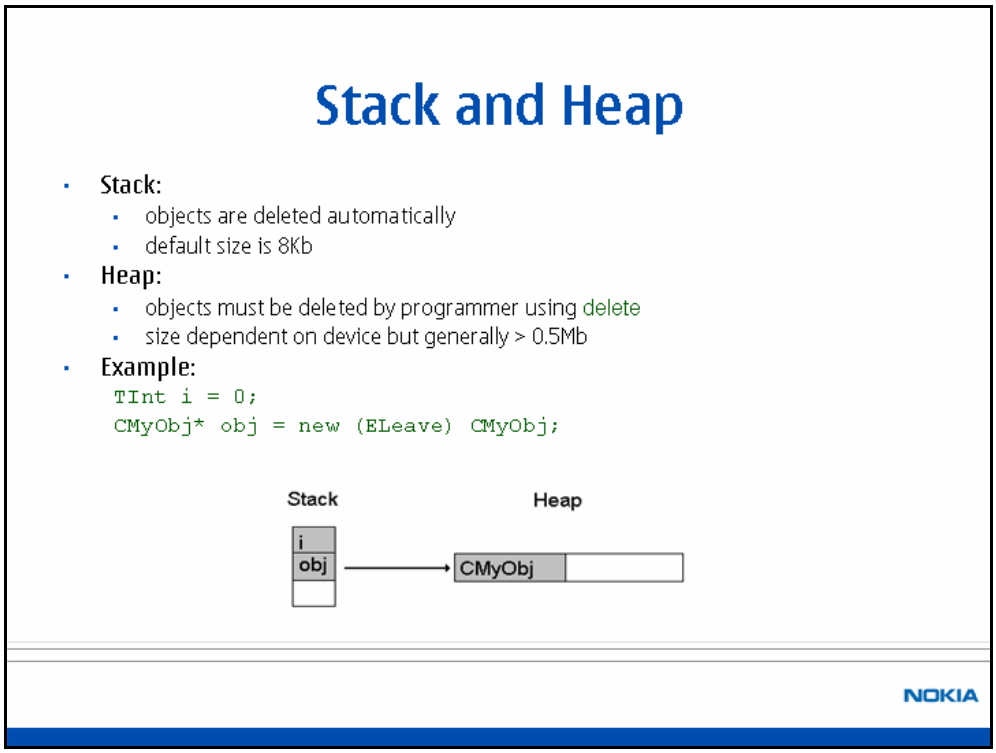
Run-time errors may occur in any application due to lack of resources: for example, a machine runs out of memory or fails to access a hardware resource. These errors are known as exceptions and it is impossible to prevent them occurring by modifying the program. Therefore, programs should be able to recover from exceptions when they occur. This is particularly important in Symbian OS, for the following reasons:

- Applications are designed to run for long periods (months or even years) without interruption or system reboot.
- Applications are designed to run on machines that have limited resources, particularly memory. Therefore, an out-of-resource error is more likely to occur than in a desktop application.

This calls for key requirements for Symbian OS developers such as:

- Program efficiently so that your program does not use RAM unnecessary.
- Release resources as soon as possible (remember resources are limited).
- Cope with out-of-memory errors. In fact you need to do that in every operation where memory is allocated.
- When out-of-memory situation arises in the middle of an operation, go back to an acceptable and stable state and make sure that you clean up all resources allocated during that operation.

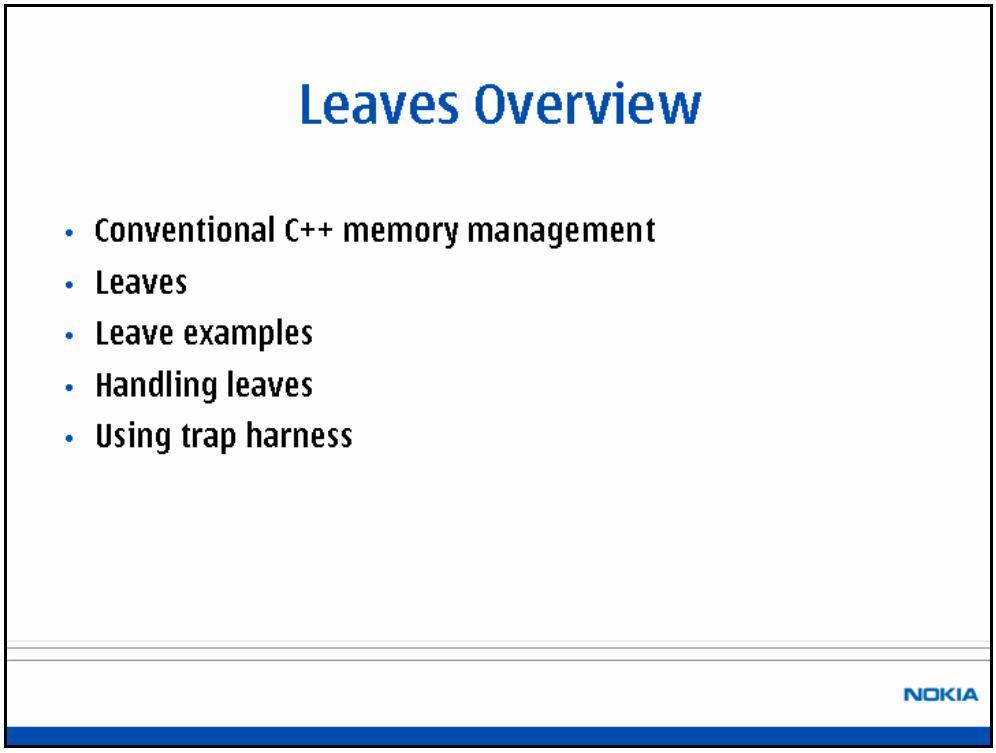
Stack and Heap



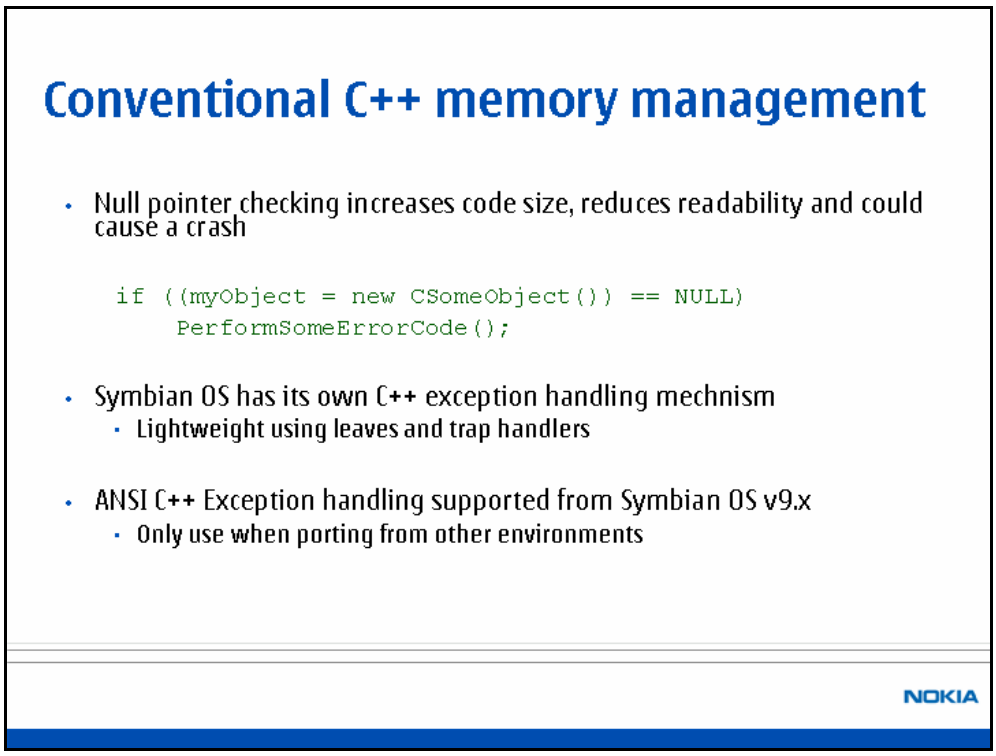
When objects are declared on the stack, they are deleted automatically when the object goes out of scope. The default stack size is 8KB, but it is possible to alter this value for Carbide.c++ projects by editing a project setting. Go to “Properties -> C/C++ Build” and select the “S60 3.0 Phone (GCCE) Release” configuration. Then select “Post Linker -> General Options” item in the “Tools settings” list. Edit the “Stack size in bytes” field to change the stack size from the default. Note that this setting only applies to phone builds and not to emulator builds.

When an object is allocated dynamically on the heap, it needs to be specifically deleted by the programmer using the `delete` keyword. If it is not a memory leak occurs. The size of heap will vary according to the memory available on a given device, but should be at least 0.5MB.

Leaves Overview



Conventional C++ memory management



We will start by covering ANSI C++ features that are not supported in Symbian OS, and memory handling techniques that are not recommended because they are inefficient.

Conventional NULL pointer checking

In a conventional C++ program, an if statement might typically be used to check for an out-of-resource error. Take a conventional C++ class, `CSomeObject`, whose definition is (note the different coding style):

```
class CSomeObject
{
public:
    CSomeObject();
    virtual ~CSomeObject();
private:
    unsigned char* m_MemPtr;
};

CSomeObject::CSomeObject()
{
    m_MemPtr = new unsigned char[100];
}

CSomeObject::~CSomeObject()
{
    delete m_MemPtr;
}
```

If this were to be allocated dynamically (i.e. on the heap) we would do the following:

```
if ((myObject = new CSomeObject()) == NULL)
```

```
    PerformSomeErrorCode();
```

There are two problems with this approach:

- It requires many extra lines of code to put such error checking round every single function that might cause an out-of-resource error. This increases code size and reduces readability.
- If a constructor fails to allocate resources, there is no way to return an error code, because constructors have no return value. The result would be an incompletely allocated object, which could cause a program crash.

Conventional C++ exception handling

ANSI C++ exception handling could be used as follows:

The `CSomeObject` constructor could be modified to do the following:

```
CSomeObject::CSomeObject()
{
    m_MemPtr = new unsigned char[100];
    if (!m_MemPtr)
        throw KOutOfMemoryException;
}
```

where `KOutOfMemoryException` is defined as:

```
const int KOutOfMemoryException = 0;
```

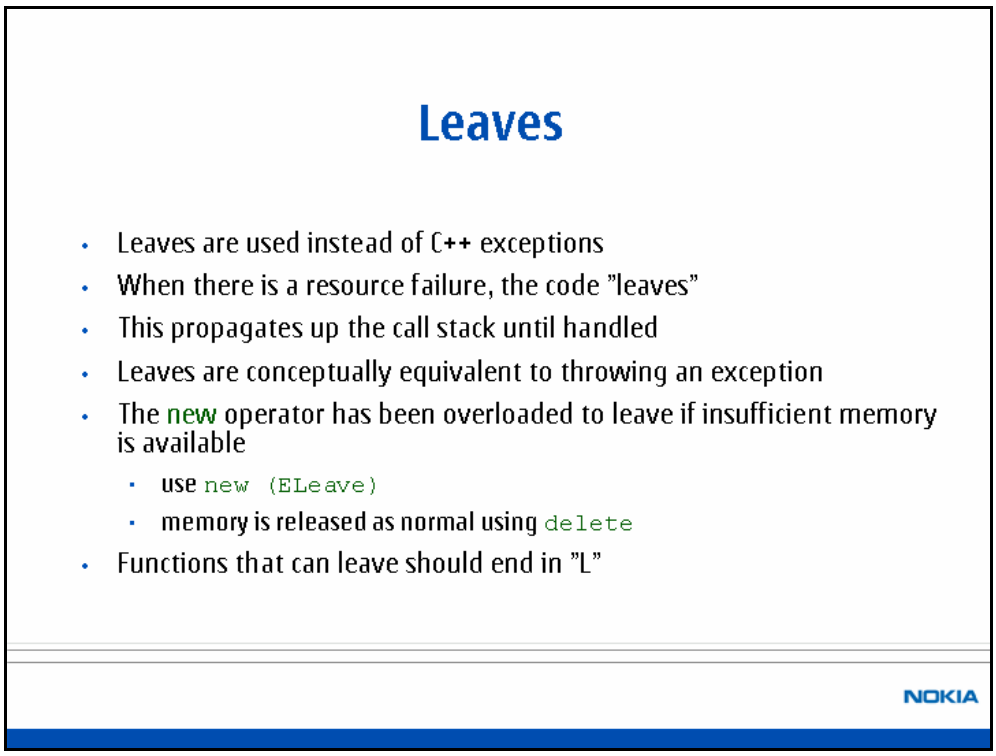
The usage of `CSomeClass` would now be:

```
try
{
    myObject = new CSomeObject;
}
catch (int e)
{
    if (KOutOfMemoryException == e)
    {
        // constructor failed to allocate resources
        // so we need to perform error handling
        PerformSomeErrorCode();
    }
}
```

Historically, ANSI C++ exception handling could not be used because it was not supported by the compiler used for device builds. Even if the compiler had supported it, early versions of the Symbian OS would have struggled to use ANSI C++ exception handling for efficiency reasons. To provide an alternative to rather awkward conventional error-checking, that would have been present with no exception handling mechanism, an alternative lightweight exception handling construct was devised using leaves and trap handlers.

Note that with Symbian OS v9.x, ANSI C++ exception handling is now supported. Although it is supported, it is recommended that standard C++ exceptions should only be used in code being ported from other environments and should not be mixed with code using the Symbian OS exception handling mechanism.

Leaves



Symbian OS introduces the concept of a leave instead of a C++ exception. If memory or resources cannot be allocated, then the code will leave. This is analogous to an exception being thrown in ANSI C++. If a leave occurs in a given function, it will occur in all the functions that call it until handled. If it isn't handled inside the application, the operating system performs some default handling.

The new operator, which is used in C++ to allocate dynamic memory, has been overridden to take a parameter, `ELeave`. When called with this parameter, new will leave if it fails to allocate the required memory. This is implemented globally and at `CBase` class level. At global level, the definition of the overridden new operator looks like the following:

```
inline TAny* operator new(TUint aSize, TLeave)
    {
    return User::AllocL(aSize);
    }
```

The above definition will be called for any dynamic allocation of T classes. `User::AllocL()` is a system static function provided by Symbian OS to allocate dynamic memory. At `CBase` level, the definition of the overridden new operator looks like the following:

```
inline TAny* CBase::operator new(TUint aSize, TLeave)
    {
    return newL(aSize);
    }
```

The above definition will be called when allocating all `CBase`-derived classes. `newL()` is a static function private to `CBase` which allocates that memory for the class and also initializes all the member data to 0.

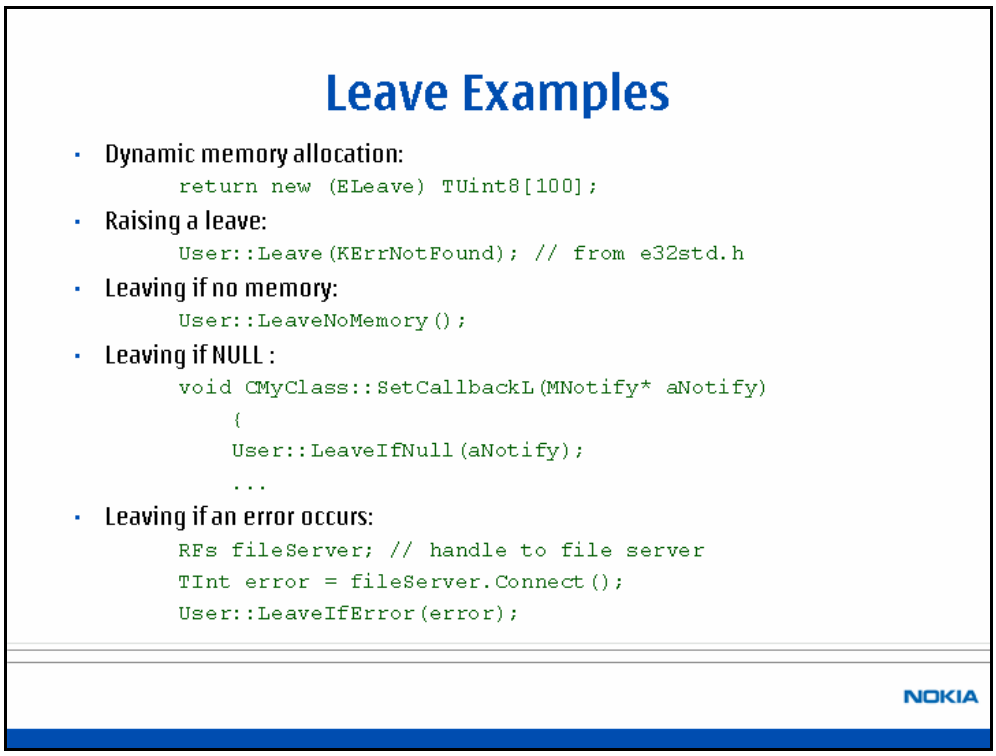
An example of the usage of the `new (ELeave)` is a follows:

```
CEikAppUi* CMyAppDocument::CreateAppUiL()
    {
    return new (ELeave) CMyAppAppUi;
    }
```

This will call the `CBase` version of new. Note that the size is passed into the new operator implicitly.

Any function that contains code that could leave should have an 'L' appended to the function name. This coding convention is necessary so that the caller of the function knows it can leave. Forgetting to put an 'L' at the end of a function does not cause a compiler error, but may introduce errors in the code because the user of such a function does not know it can leave.

Leave Examples



The first example on the slide shows how to allocate memory so that the code will leave if the allocation fails.

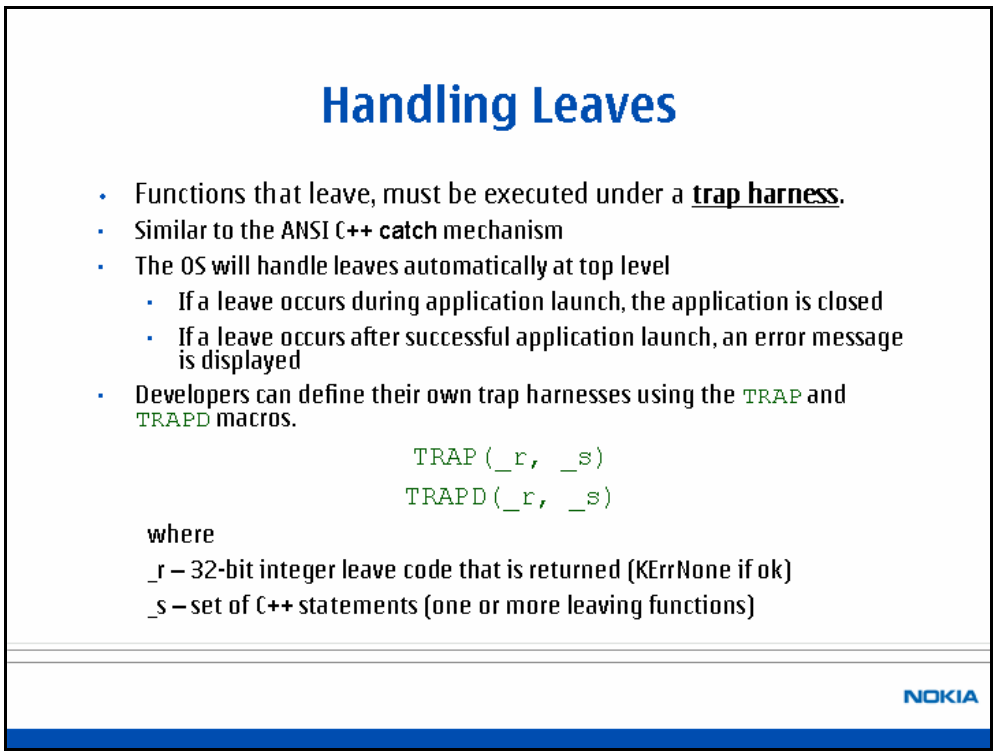
The second example shows how to leave using the `User::Leave()` method. This function takes a single parameter, which specifies a return code. Application defined values can be used when raising leaves. The set of system error codes, starting with `KErrNone` (meaning no error) can be found in the header file `e32std.h`.

The third code example shows how to leave in the case of no memory left..

The fourth code example shows a useful way of NULL pointer checking. This is often used to check that function input parameters are not NULL.

The final example shows how to leave if an integer error code is returned from a system API. In the example a connection to the file server is attempted (servers will be covered in a later section). If it fails a system error code will be returned and a leave of this error value is raised. Note that it is generally good practice in application code to leave rather than return an error.

Handling Leaves



The operating system provides its own mechanism for handling leaves which is analogous to the ANSI C++ catch mechanism.

The OS performs default handling of leaves. The timing of the leave can affect the outcome. This is intuitive because if there is not enough memory or resources to launch an application successfully, then there is no point in continuing with the application so it may as well be closed. However, once the application has been successfully launched, the user may invoke different functionality through menu items and key presses. One operation may require more memory than another, so if a memory heavy operation failed, it may be useful to allow the user to try a more lightweight operation. Therefore the application is not exited.

It is possible for application developers to handle leaves themselves by using the `TRAP` and `TRAPD` macros. For example an application may need to reset its state following a leave.

`TRAP(_r, _s)` - Executes the set of C++ statements `_s` under a trap harness.

`_r` is a `TInt` leave code. If any of the C++ statements `_s` leaves, then the leave code is returned in `_r`, otherwise `_r` is set to `KErrNone`. `_r` must have been declared prior to using `TRAP`, or alternatively it is not declared in the case of `TRAPD`.

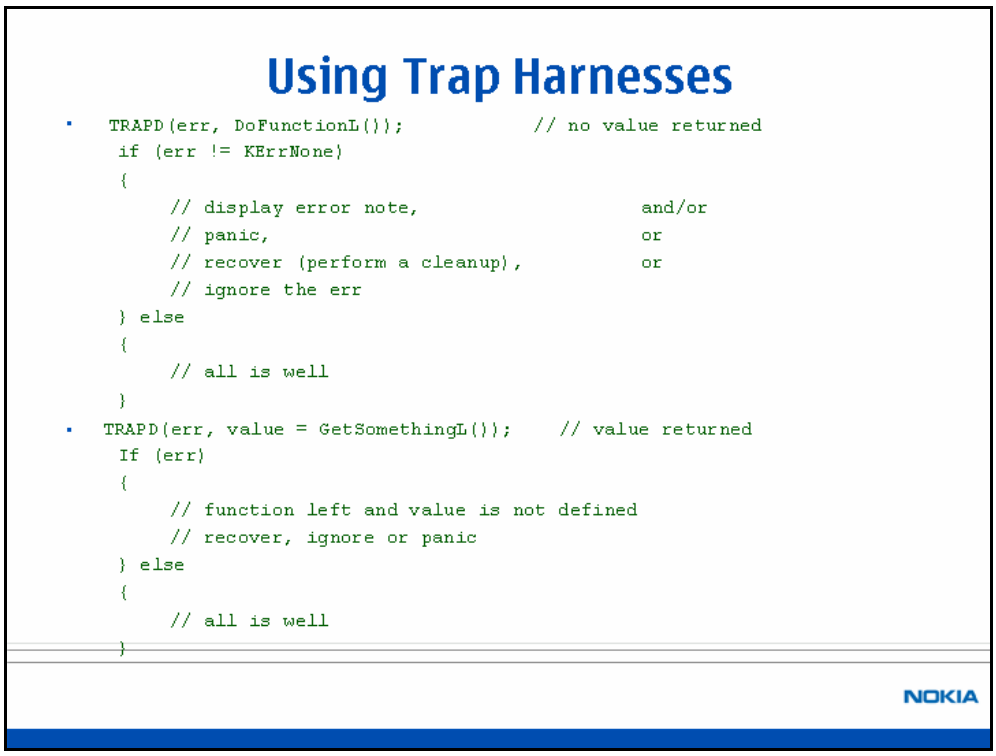
`_s` can consist of multiple C++ statements; in theory, _s can consist of any legal C++ code but in practice, such statements consist of simple function calls.

Whenever a leave occurs within code executed inside the trap harness, program control returns immediately to the trap harness. The macro then returns an error code, which can be used by the calling function. The error code value is the same as the error code raised by the leave.

The existence of `TRAPD` is just for convenience – it saves a line of code.

Note that macros are preprocessor directives that cause code substitution at compile-time. The advantage of them is that they can reduce code size over function calls. However, there is a disadvantage in that they are not as type-safe as function calls. So, trap harnesses should only be used as the exception rather than the rule as they increase code size. Most of the time the operating system provides sufficient error handling as it displays an error dialog.

Using Trap Harnesses



Typically after a `TRAP`, a function checks the leave variable to test whether processing returned normally or by a leave, and acts appropriately. Special mechanisms discussed later are provided to handle cleanup after the exception.

Two use cases are shown on the slide. In the first one the leaving function does not return a value. If the function has left, then typical scenarios are to display a message note and/or to try to recover by performing some cleanup or ignore the error or panic (explained later). In the second use case, the function returns a value. Note that if the function leaves, the returned value will be undefined, so do not try to use it.

The following is an example of multiple functions in a trap harness where each of the functions could leave:

```
TInt error;
TRAP(error,
    {
    DoFunction1L();
    DoFunction2L();
    }
);

if (error != KErrNone)
    {
    // error handling
    }
```

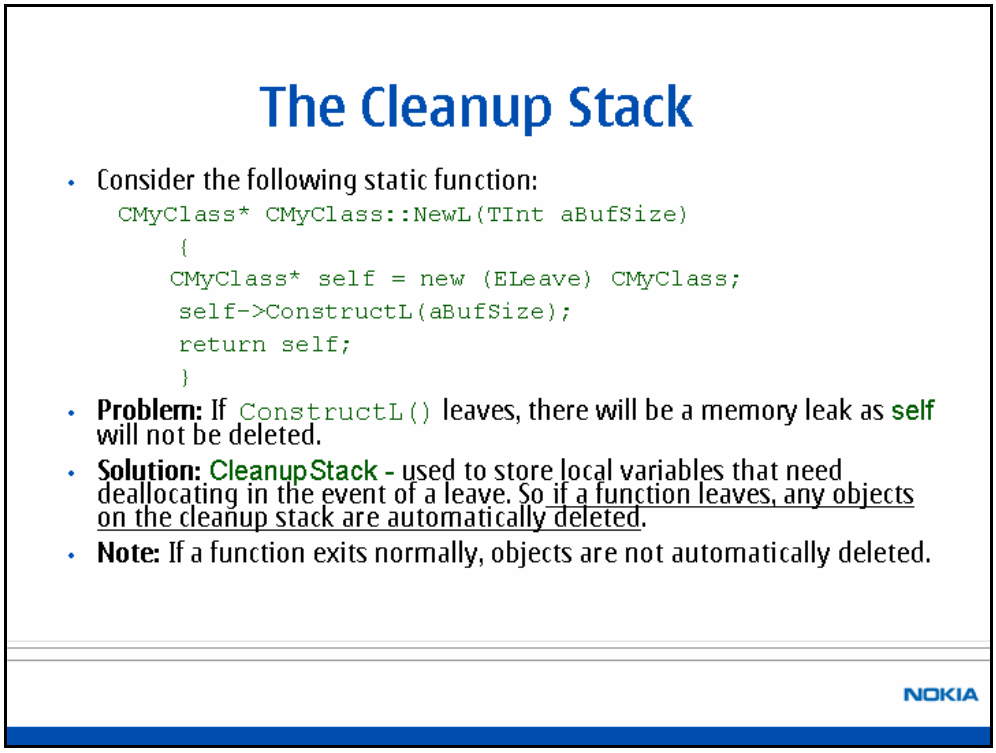
If either of the functions leave then the error will be picked up in error. This can be convenient if you are not interested in which particular step caused the error. You can, of course, have two trap harnesses put around each function but that comes at a cost, so avoid it as a general rule of thumb. On the other hand, for readability, avoid having more

than few statements in one trap harness. If you need to, then create a function that encapsulates all the statements and call that function in the trap harness.

Another example that deals with the different possible error codes returned by a leaving function is:

```
TRAPD( err, LaunchAppL() );

if  (err == KErrNotFound)
    {
    DisplayNotFoundNote();
    …
    }
else if (err == KErrNotReady)
    {
    DisplayNotReadyNote();
    …
    }
else if (err != KErrNone)
    {
    DisplayGeneralErrorNote();
    …
    }
else
    {
    // all is well
    }
```

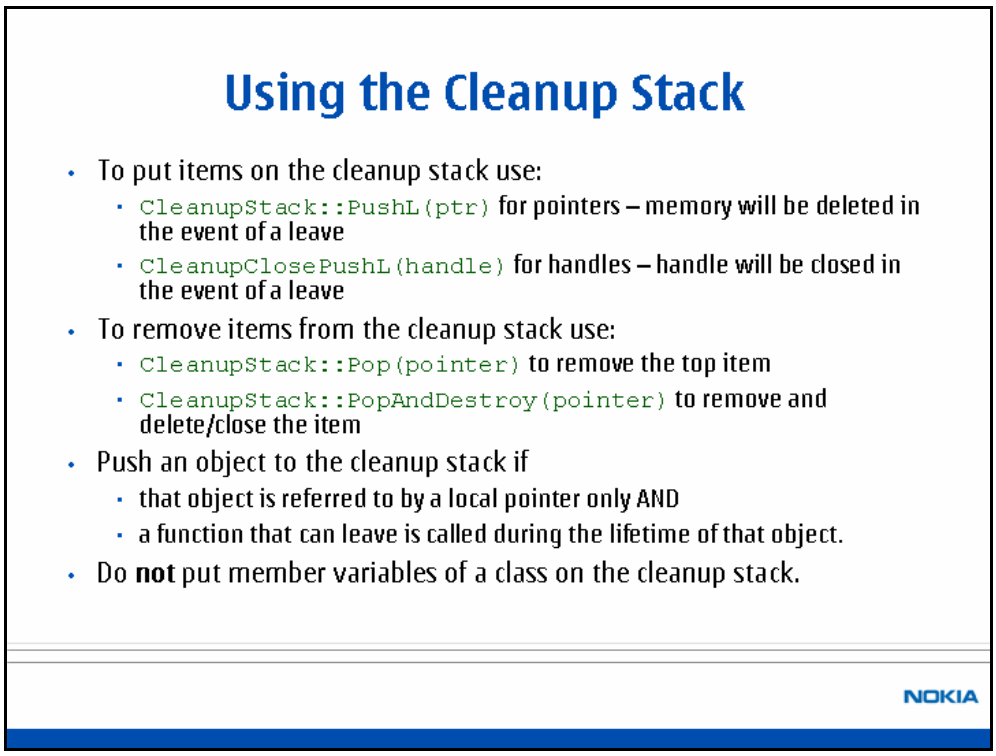
The Cleanup Stack



The code example on the slide, illustrates a function that has a local variable that owns dynamically allocated memory (until the pointer is returned to the caller). As `ConstructL()` ends with an uppercase L, developers know that it could leave. If it does leave, the memory address pointed to by `self` will become unreferenced. There would be no way of deallocating the memory and so a memory leak would arise.

To get around this problem, Symbian OS uses a global cleanup stack. It is accessed via static functions of the class `CleanupStack` and references all the variables that would become unreferenced in the event of a leave. The operating system will clean up the memory/resources associated with all variables on the cleanup stack when an unhandled leave occurs.

Using the Cleanup Stack



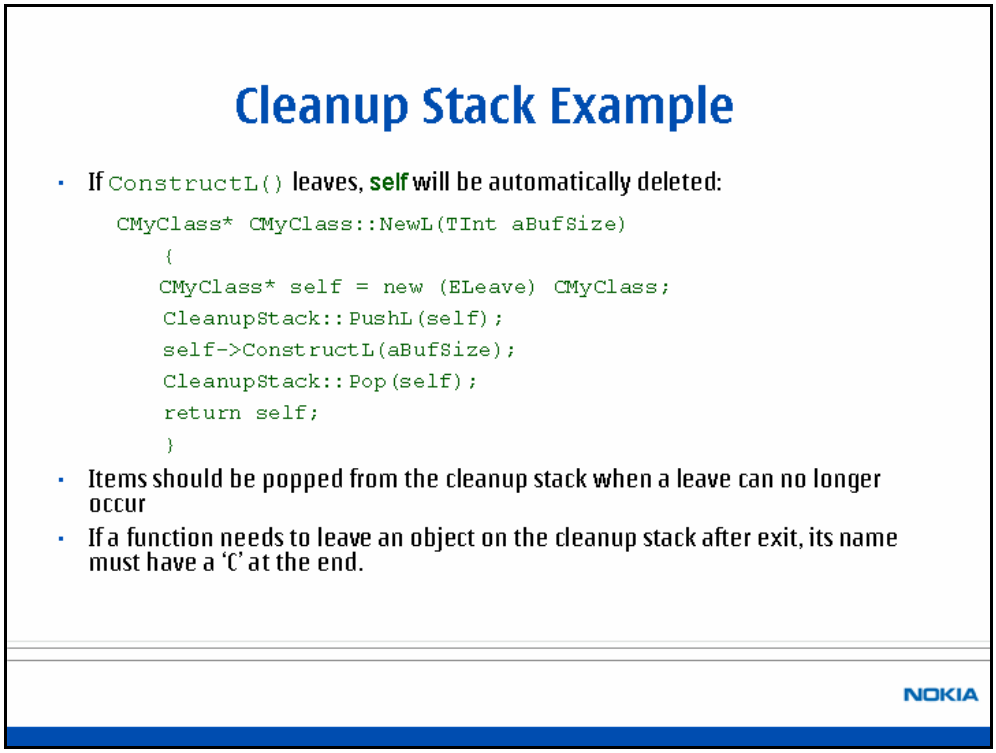
The slide shows the APIs used to place items on the cleanup stack and remove them from it. As well as pointers, it is possible to put open handles on the cleanup stack. In the event of a leave, all handles on the cleanup stack will be closed. An example of this will be shown in the client/server module.

If items are not popped from the stack after they have been closed or deleted, an unrelated leave later on will cause a panic to occur. Similarly, member data should not be placed on the cleanup stack, since it will be deleted/closed twice after a leave (once by the operating system, then again by the class that owns it when it is destructed), which will cause a panic.

In summary, if a function can leave check what happens.

- If it leaves, are all objects on the heap referred to only by local pointers on the cleanup stack?
- If it does not leave, have you done a proper cleanup yourself?

Cleanup Stack Example



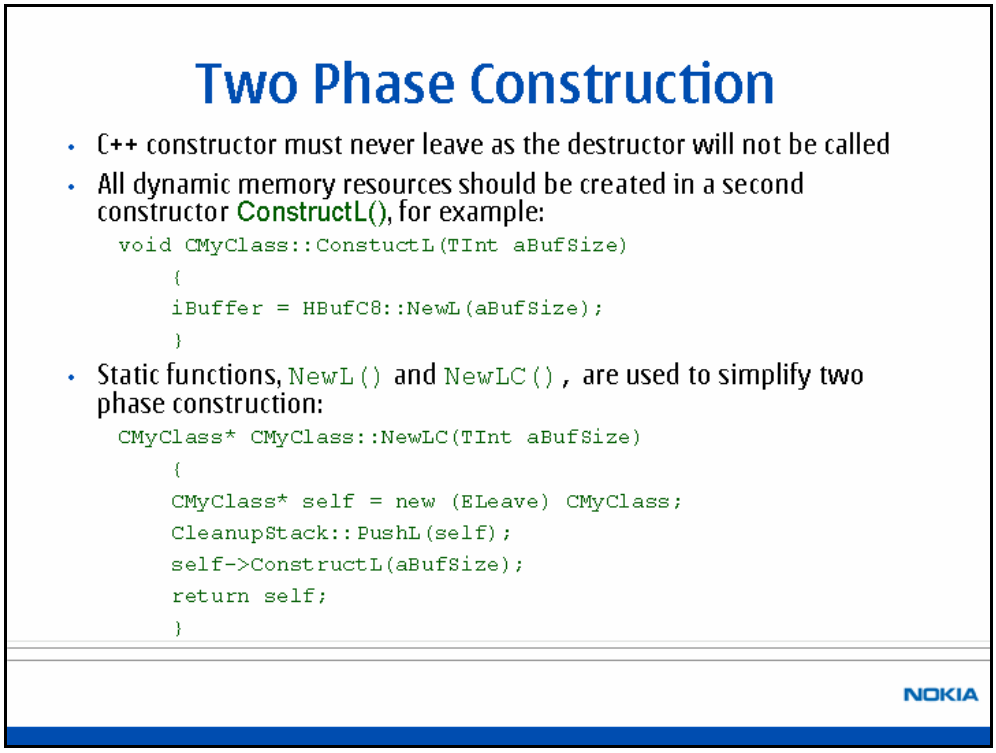
This shows the earlier example corrected with the addition of two cleanup stack calls. Note that once no further leaves can occur within the function, self is popped. If self was not being returned to the caller and was instead deleted, `CleanupStack::PopAndDestroy(pointer)` could be called instead of `CleanupStack::Pop(pointer)`. Supplying the optional "pointer" parameter allows the kernel to check that the item being removed from the cleanup stack is the one expected.

To remove multiple items from the cleanup stack, `CleanupStack::Pop(3, pointer)` can be used. In this case 3 items are being removed from the stack. Again the optional "pointer" parameter allows the kernel to check that the final item removed from the stack is the one expected.

Sometimes an object may be left on the cleanup stack in a function. If a function does need to leave an object on the stack after exit, its name must have a 'C' at the end. So the caller will know that he is responsible for that object.

The type of static function shown in the example is a very common occurrence in Symbian OS development. It provides two-phase construction. The next slide will explain this concept.

Two Phase Construction



Another important Symbian OS concept is two-phase construction. The need for this arises if a constructor needs to allocate resources, for example dynamically allocating member data. For example in the following hypothetical case:

```
CMyAppAppUi::CMyAppAppUi()
    {
    …
    iAppContainer = new (ELeave) CMyAppContainer;
    …
    }
```

If the memory for `CMyAppAppUI` had been successfully allocated in `CMyAppDocument::CreateAppUiL()`, and the constructor leaves allocating `iAppContainer`, the pointer to the memory that was allocated for `CMyAppAppUi` will be lost, causing a memory leak. The solution to this problem is to have another constructor, known as the second phase constructor that can leave:

```
void CMyAppAppUi::ConstructL()
    {
    …
    iAppContainer = new (ELeave) CMyAppContainer;
    …
    }
```

`NewL()` and `NewLC()`

These static functions perform both construction phases in one go. Providing `NewL()` or `NewLC()` functions as part of a class makes it easier to use that class. The code example on the previous slide shows an example of a `NewL()` function. The automatic self is put on the cleanup stack in case `self->ConstructL()` leaves. The `NewLC()` function seen

on this slide, does not remove the constructed object from the cleanup stack, the calling code should call `CleanupStack::PopAndDestroy()` when it has finished using the constructed object (assuming it was using it as a local variable). `NewLC()` would typically be used where one function contains a series of automatic variables (which point to heap memory), to save pushing each one onto the cleanup stack.

Coding guidelines

It is recommend that all user-defined 'C' classes should:

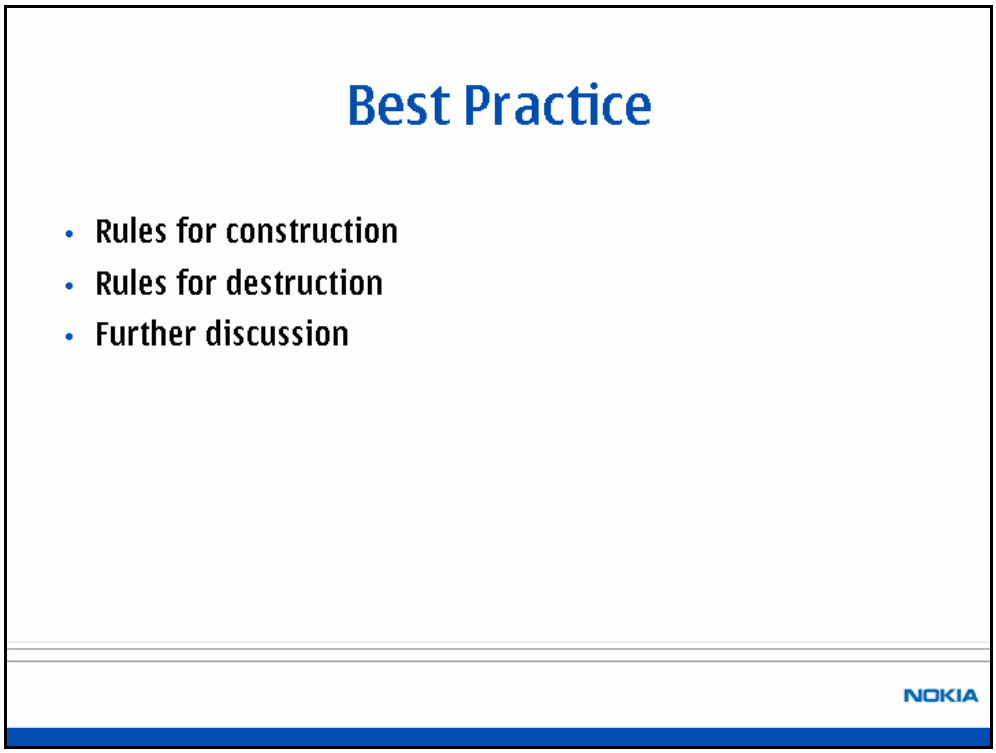
- Define, `New` and `NewLC` functions. These should be defined as `public` and `static`.
- Define a `private ConstructL()` and C++ constructors.

Following this pattern makes classes easier to read and maintain. Making the C++ constructor `private` means that users of the class cannot attempt to call functions without the `ConstructL()` function having been called first.

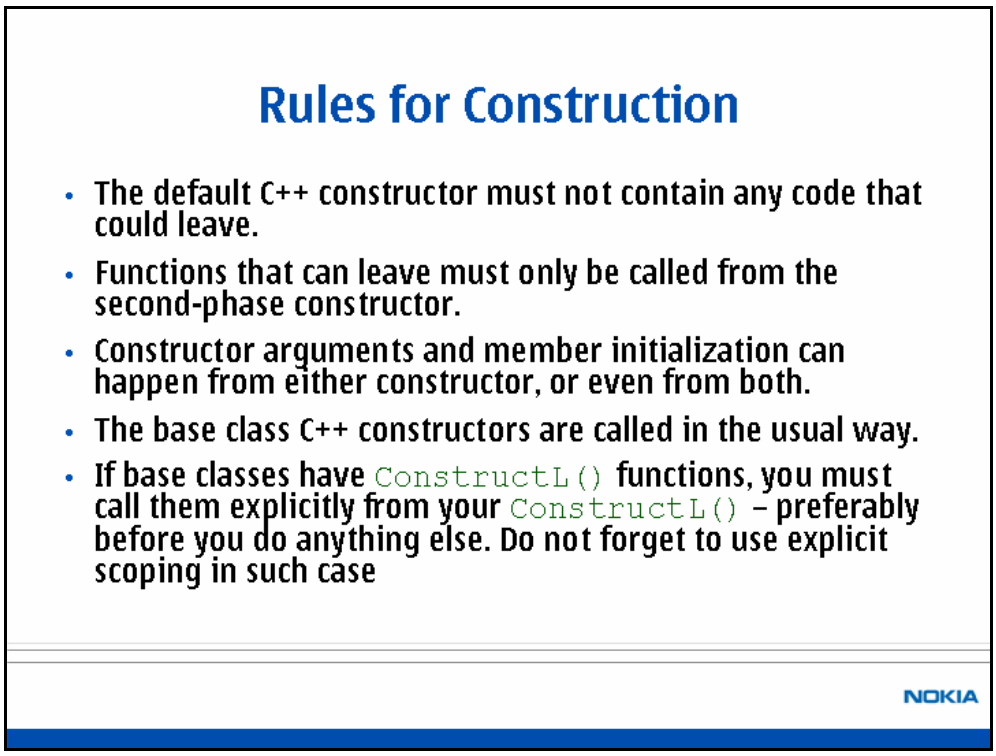
In the rare occasions that a 'C' class is intended to be derived from, and is not intended to be concrete, then:

- Do not define `NewL()` or `NewLC()`.
- Define a private/protected `BaseConstructL()` function and C++ constructors.

Best Practice



Rules for Construction



Rule 1:

The default first-phase C++ constructor must not contain any code that could leave. For example:

```
CParser::CParser(CFont& aFont) : iFont(&aFont)
    {
    }
```

Rule 2:

Functions that can leave must only be called from the second-phase constructor. For example:

```
void CParser::ConstructL(CResourceReader& aReader)
    {
    iParserInfo = CParserFormatInfo::NewL(aReader);
    iTextLines = new (ELeave) CDesCArrayFlat(1);
    iTotalText = HBufC::NewL(0);
    }
```

Rule 3:

Constructor arguments and member initialization can happen from either constructor, or even from both.The base class C++ constructors are called in the usual way. For example:

```
CParserTopDown::CParserTopDown(CFont& aFont)
:   CParser(aFont),
    iSplitStatus(ENormal)
    {
```

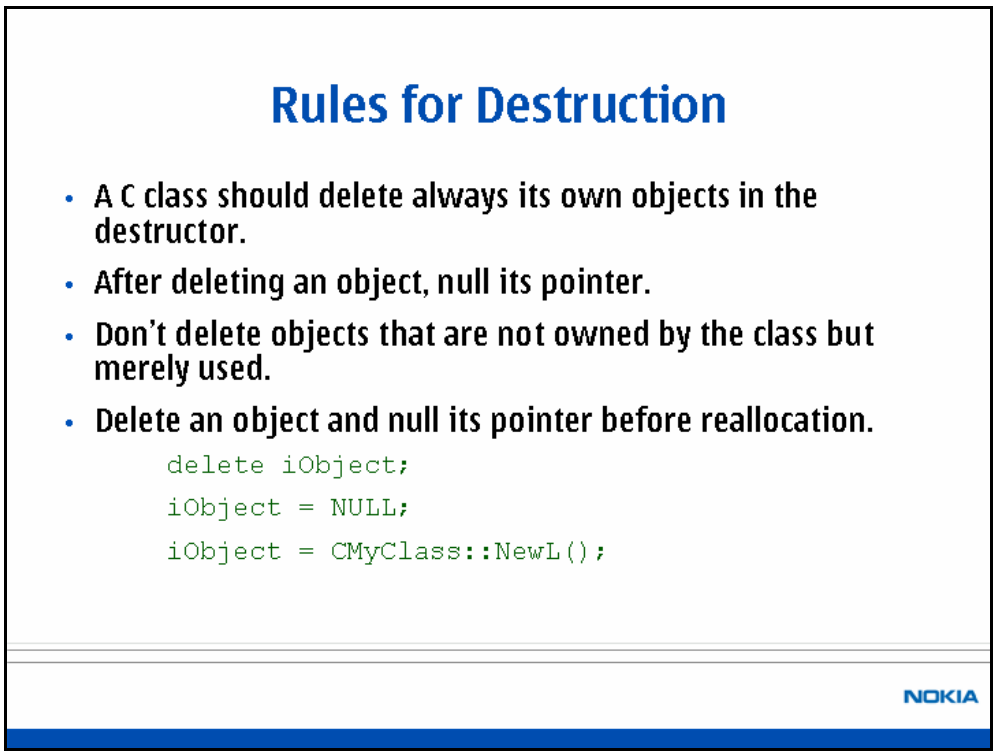
```
    }
```

Rule 4:

If base classes have `ConstructL()` functions, you must call them explicitly from your `ConstructL()` – preferably before you do anything else. Do not forget to use explicit scoping in such cases. For example:

```
void CParserTopDown::ConstructL(CCuiResourceReader& aReader)
    {
    CParser::ConstructL(aReader);
    }
```

Rules for Destruction



Rule 1:

A C class should always delete its own objects in the destructor. Note that it is OK to delete a pointer that is NULL, so there is no need to write:

```
if ( iPointer )
    {
    delete iPointer;
    }
```

Rule 2:

After deleting an object, immediately set its pointer to NULL.

```
delete iObjectPointer;
iObjectPointer = NULL;
```

Nulling the pointer is always a good practice and sometimes it is crucial as will be shown in another example below.

Rule 3:

Do not delete objects that are not owned by the class but merely used. Imagine a class which is passed upon construction a pointer to an observer interface. For example:

```
class CMyClass: public CBase
    {
    public:
    …
        void SetObserver(CMyClassObserver* aObserver)
```

```
    …
    private:
        CMyClassObserver* iObserver; // we do not own this
    …
    };
```

In the destructor, set the pointer to NULL but do not delete.

```
iObserver = NULL;
```

Rule 4:

Always delete an object and null its pointer before reallocation.

Imagine a class `CMyClass` that has an object created on the heap that needs updating. For example:

```
CMyClass::UpdateObjectL()
    {
    iObject = CMyObject::NewL();
    }
```

If you call the function `UpdateObject()` function twice, then you will allocate memory twice, which will result in a memory leak. Lets fix the memory leak. Our function now looks like the following:

```
CMyClass::UpdateObjectL()
    {
    delete iObject;
    iObject = CMyObject::NewL();
    }
```

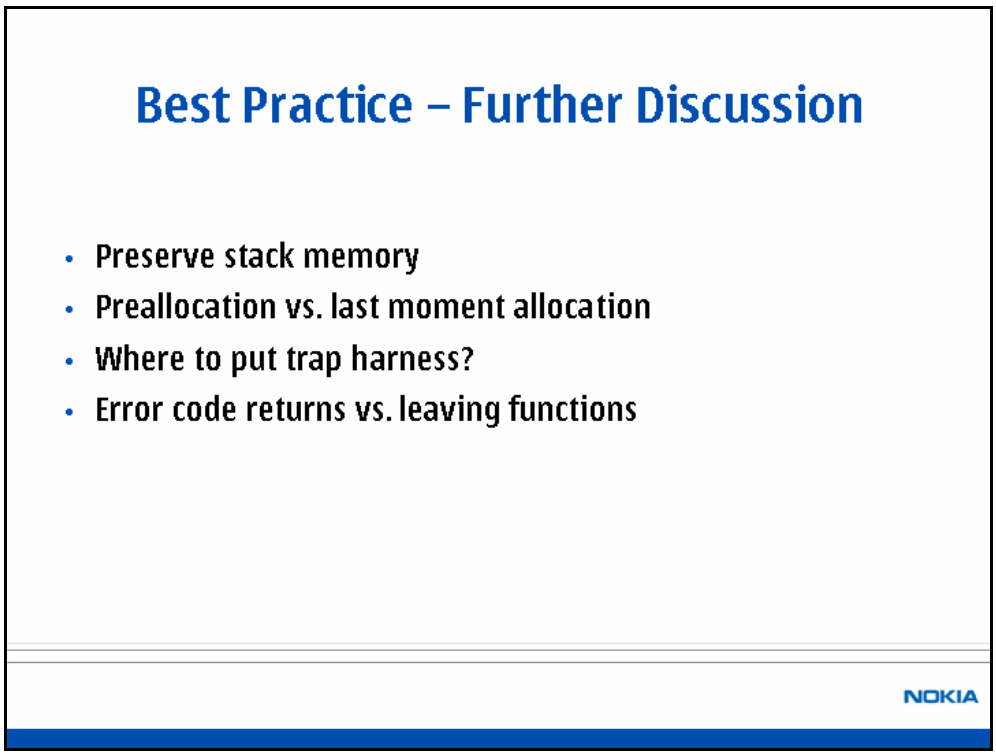
There is still a problem in the code above. If the object creation fails, that is the `NewL()` leaves, `iObject` is still pointing to the previously allocated object and the chances are you will try to delete it again (perhaps in the destructor) causing double deletion and a program crash as a result. Having cleared this we can now write the correct version of our function:

```
CMyClass::UpdateObjectL()
    {
    delete iObject;
    iObject = NULL; // nulling the pointer is crucial
    iObject = CMyObject::NewL();
    }
```

Sometimes the deletion of the object and cleaning of the pointer might happen somewhere else in the code (in another function). The following check is a simple way to ensure the pointer does not already point to anything before reallocation occurs:

```
ASSERT(iObject == NULL);
```

Best Practice – Further Discussion



Preserve stack memory

Each thread has only 8K. So use it wisely and follow these guidelines:

- Pass function parameters by reference instead of by value (except for basic types).
- Create large objects or arrays on the heap rather than the stack.
- Minimise the lifetime of local variables by appropriately scoping them.

Preallocation vs. last moment allocation

Although the general rule is to allocate resources just before their use and release them as soon as possible in view of the limited memory, sometimes it is more advantageous to do a preallocation. Preallocation can save processing time and will have the resource ready for use even if at a later stage an out-of-memory situation arises.

For example, consider a drawing function that reads a text string from a resource file, draws this text on the screen and frees the resource (each time the function is called). Such a function will be slower than a function that just displays the text on the screen, with the text preallocated in an associated constructor and freed in a destructor. In addition, once the text has been preallocated the function will still work in out-of-memory situations.

Where to put trap harness?

The most basic option is to rely on the top-level trap harness provided as part of the application framework for all GUI programs. If a leave occurs, and it is not handled by any explicitly coded harness, the framework displays an error message corresponding to the leave code.

For some applications, the unit of recovery may be associated with part of the processing of a particular command — a much finer-grain approach.

Coding a coarse-grain unit of recovery has the advantage of only one trap harness and recovery code, but the disadvantage that recovery code may be general and complex, and the danger that a small error leads to catastrophic results for a user (for example if not enough memory to apply bold formatting resulted in termination of the word processor with loss of data!).

Coding too fine-grain units of recovery results in many trap harnesses, lots of recovery code which may require individual attention in each case, and potentially significantly increased code size.

The correct choice is application-specific. For large applications, there may be a single course-grain unit of recovery, with other harnesses in particular places.

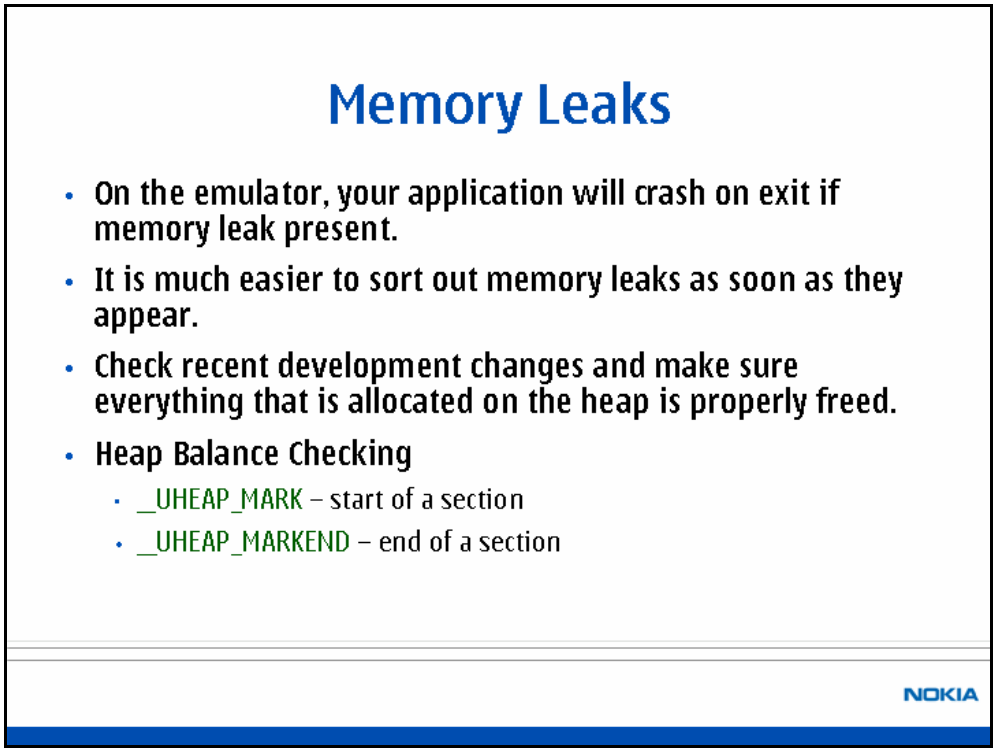
Error code returns vs. leaving functions

Exception handling and cleanup techniques is one approach to handle environment and user errors. Another approach would be to detect problems before an action is performed then return an integer value other than `KErrNone`. This method is used mainly in 'R' type classes.

For example, the file server provides the functions `Connect()` and `Delete()` which return a `TInt` error code, including `KErrNone` if the functions performs successfully. This means that you have to check the error explicitly using `User::LeaveIfError()`.

```
void DeleteFileL(const TDesc& aFileName)
    {
    RFs fs; // file server
    CleanupClosePushL(fs);
    User::LeaveIfError(fs.Connect());
    User::LeaveIfError(fs.Delete(aFileName));
    CleanupStack::PopAndDestroy(&fs);
    }
```

Memory Leaks



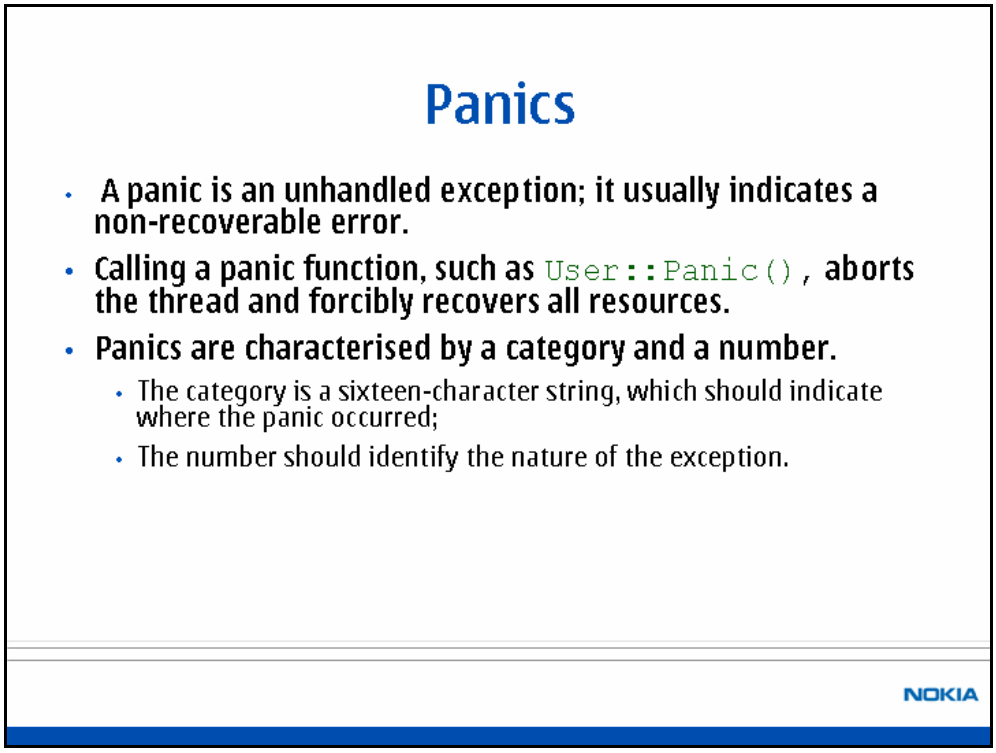
Ok. You have understood the importance of memory management and are determined to follow the rules strictly. Yet you were not totally focused during the development of your application and it has got a memory leak. How would you know that? And once you know you have a memory leak what do you do? These questions are answered below.

An application running in debug mode will already include code to raise a panic if on exit a memory leak is found. A general rule of thumb is to sort out a memory leak as soon as it is detected. You will know what development changes you have made since the last successful run of the application and somewhere within these changes hides your memory leak. Check all your allocations on the heap and make sure that all memory is freed properly. If you still cannot find where the problem is try the following method for heap balance checking to localise it.

Heap balance checking

Use the macros `__UHEAP_MARK` and `__UHEAP_MARKEND` to enclose a section of code where you need to check whether the heap is balanced. If the heap does not have the same number of cells allocated at the point of call of `__UHEAP_MARKEND` as when `__UHEAP_MARK` was called you will get a panic. This is an indication that you have a memory leak taking place in-between. The macros can be nested if desired.

Panics



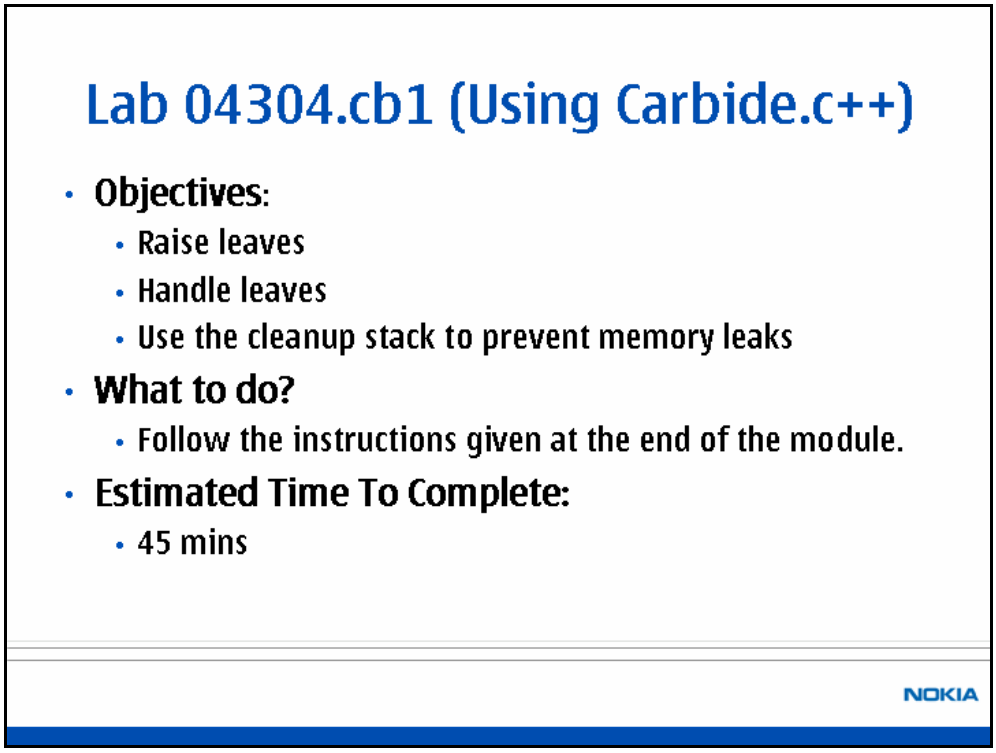
This section is not about memory management however it is related to it in the sense that it provides a way of coping with another type of extreme situation.

Errors can be broadly divided into three types:

- Program errors, such as an attempt to access an element beyond the bounds of an array or buffer.
- Environment errors, such as insufficient memory, insufficient disk space, or other missing resources.
- User errors, such as an attempt to enter bad data in a dialog, an invalid action, or bad syntax in a source file.

You can use trap harness and cleanup techniques to handle errors of the second and third type. The type of errors that cannot be handled in this way are programming errors. We do not want the application to recover from such errors. Symbian OS can be made to panic your application as soon as such an error is detected and can provide meaningful diagnostic information for you to use. The panic function is `User::Panic()` which takes a panic category string (<= 16 characters) and an error code. On a real machine your application will get closed and you will get a dialog citing the process name, the panic category and the panic number. Therefore, it is important to make the category string and the error code specific.

Lab 04304.cb1 (Using Carbide.c++)



Lab 04304.cb1

Overview

Title:	Raising and Handling Leaves and Using the Cleanup Stack to Prevent Memory Leaks		
Overview:	This lab aims to make the programmer aware of memory considerations when writing Symbian applications.		
Objectives:	To understand: • What happens when a Leave occurs. • When to handle the Leave to take corrective action. • When and why the Cleanup Stack should be used.		
Compatible IDE(s):	Carbide.c++ v1.2		
Compatible SDK(s):	S60 3rd Edition, S60 3rd Edition (MR), S60 3rd Edition FP1		
App. Type:	Standard GUI	**App. Name:**	S60MemoryLab.exe
Starter Code Provided:	Yes	**Solution Code Provided:**	Yes
Estimated Time To Complete:	45 minutes		

Lab Instructions

Exercise 1 – Preliminary Steps

Task 1.1 – Checks

1. Ensure that a compatible SDK has been installed. A list of compatible SDKs to use with this lab is given in the table in the Introduction.
2. Ensure that a compatible IDE has been installed. A list of compatible IDEs to use with this lab is given in the table in the Introduction.
3. Ensure that the lab files are located in the folder `C:\Labs\Lab_04304.cb1\`. If this is not the case please refer to the setup guide for details of how to obtain them.

Task 1.2 – Make sure the emulator is not in text shell mode

1. Open Windows Explorer.
2. Navigate to the file named epoc.ini, located in the `<EPOCROOT>\epoc32\data` folder. Here, `<EPOCROOT>` is a placeholder for the root folder location of your SDK.
3. Open the epoc.ini file using Notepad.
4. Check that the file does not contain the “textshell” statement. If the statement is present in the file, then either delete the entire line containing the statement or comment the statement out. The latter is done by inserting a # symbol at the start of the line containing the statement.

5. Press 'Ctrl' + 'S' to save the contents of the file (if you altered it) and close Notepad.
6. Close Windows Explorer.

Task 1.3 – Carbide Setup

1. Launch the Carbide.c++ IDE. Depending on your Carbide.c++ settings the IDE will either display the "Workspace Launcher" dialog or open the last used workspace. If the latter occurs open the "Workspace Launcher" dialog by selecting the "File -> Switch Workspace..." menu item.
2. In the workspace launcher window, create a new workspace by typing `C:\Labs\<Workspace>` in the Workspace field. Here `<Workspace>` is a placeholder for a folder name of you choosing that does not already exist and does not contain spaces.
3. Click the OK button.
4. Close the Welcome window after it appears when the workspace is opened.

Task 1.4 – Importing the project

8. In the Carbide.c++ IDE, select the "File > Import" menu item. The Import dialog appears.
9. Select "Symbian OS bld.inf file" from the "Symbian OS" subfolder as the import source and click the Next button. A new screen appears on the Import dialog.
10. Enter `C:\Labs\Lab_04304.cb1\starter\group\bld.inf` in the "bld.inf file" field. Alternatively, click the "Browse..." button and then navigate to and select the correct file via the dialog. When this is done a list of configurations appears in the "SDKs and Build Configurations pane".
11. Expand the list of configurations for your installed SDK and tick the checkbox next to the "Emulator Debug" configuration. (Note that other configurations may be chosen in addition to this one, but they are not used in this lab.) Then select the "Next" button.
12. Ensure the MMP file and all the make files are selected on the "MMP Selection" screen. Then select the Next button. The Project Properties screen appears.
13. Change the name in the Project field to "S60MemoryLab" (without the quotes) and click the Finish button to complete the import.
14. The Import dialog closes and a new project called "S60MemoryLab" appears in the "C/C++ Projects" view.

Exercise 2 - Raising and Handling Leaves

Task 2.1 – Running the application when no Leaves occur

1. Build the application for the Emulator Debug configuration of the SDK you are using by right clicking on the S60MemoryLab project node in "C/C++ Projects" view and select the "Build Project" or "Rebuild Project" menu item. The project should compile successfully with no errors.
2. Right click on the S60MemoryLab project node in "C/C++ Projects" view and select the "Run As > Run Symbian OS Application" menu item. The emulator will launch.
3. Follow the instructions for this step that are specific to the SDK you are using:

   S60 3rd Edition SDK and S60 3rd Edition MR SDK

   After the emulator has booted the Applications menu is shown. The S60MemoryLab

application is located in the “Installat.” folder. Navigate to this folder using the navigation keys and open it using the selection key.

S60 3rd Edition FP1 SDK

After the emulator has booted the Standby application is shown. Switch to the Applications menu by selecting the Applications key. The S60MemoryLab application is located in the “Installed” folder. Navigate to this folder using the navigation keys and open it using the selection key.

4. Navigate to the S60MemoryLab application, again using the navigation keys. Figure 5 shows a screenshot of the emulator after this is done.

Figure 5 Screenshot of the emulator after navigating to the S60MemoryLab application

5. Launch the S60MemoryLab application using the selection key. Figure 2 shows a screenshot of the emulator after the application has been launched.

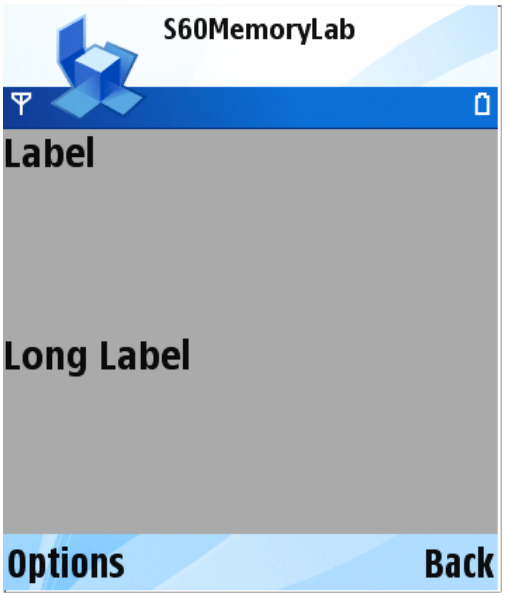


Figure 6 Screenshot of the emulator after launching the S60MemoryLab application

6. Select the Options softkey with the mouse pointer, or press F1 on the keyboard. The options menu is displayed as shown in Figure 3.

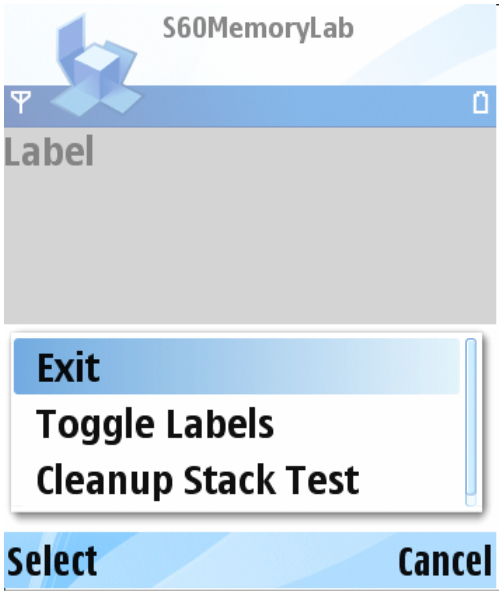


Figure 7 Screen shot of emulator window after selecting the options softkey

7. Select the "Toggle Labels" menu item using the Navigation keys. The screen shown in Figure 4 should be displayed.

Figure 8 Screen shot of emulator window after the "Toggle Labels" menu option has been selected

8. Press Back to exit the application. The Applications menu is displayed.
9. Close the emulator.

Task 2.2 – Running the application with an unhandled Leave before application construction

This task illustrates what happens when the S60MemoryLab application is modified to make an unhandled leave occur before all the application objects have been fully constructed.

1. Double click on the source file, `S60MemoryLabContainer.cpp`, located in the `/S60MemoryLab/src` folder in the "C/C++ Projects" view. A text editor window is opened showing the contents of the file.
2. Uncomment the line `User::Leave(KErrCancel);` in the function `CS60MemoryLabContainer::ConstructL()`.
3. Save the changes made to the file by selecting the "File -> Save" menu item.
4. Rebuild the application, launch the emulator and navigate to and open the S60MemoryLab application. The application leaves before construction has completed. The user is returned to the applications menu. Note that no message is displayed to the user in this case.
5. Close the emulator.
6. Undo the source code edit performed in this task by commenting out the line `User::Leave(KErrCancel);` in the function `CS60MemoryLabContainer::ConstructL()`.
7. Save the changes made to the file.

Task 2.3 – Running the application with an unhandled Leave after application construction

This task illustrates what happens when an unhandled leave occurs after an application has been successfully constructed. Follow the steps below:

6. In the text editor window showing the contents of the `S60MemoryLabContainer.cpp` file, uncomment the line `User::Leave(KErrNoMemory);` in the function `CS60MemoryLabContainer::ToggleLabelsL()`. This simulates the 2nd `SetTextL()` call leaving.
7. Save the changes made to the file.
8. Rebuild the application, launch the emulator and navigate to and open the S60MemoryLab application.
9. Once the application is open, select the Options softkey to bring up the options menu.
10. Select the "Toggle Labels" menu item. An information note is displayed as shown in Figure 9.

Figure 9 Screen shot of the emulator showing the information note

11. Once the information note disappears the screen shown in Figure 10 is displayed.

Figure 10 Screen shot of the emulator after the information note has disappeared

Note that despite the unhandled leave, the user remains within the application. Also note that only one of the labels has changed and the application is left in an invalid state and it is not possible to toggle back to the original state.

12. Exit the application by pressing the Back softkey.
13. Close the emulator.

Task 2.4 – Handling a Leave

This task shows how to handle the above Leave so that the application's state can be reset to a valid state.

1. In the text editor window showing the contents of the `S60MemoryLabContainer.cpp`

file, add a Trap harness around the label setting code in the `else` branch of the `CS60MemoryLabContainer::ToggleLabelsL()` function. Change the code below the “Edit 3” comment to the following:

```
TRAPD(
    error,
    iTopLabel->SetTextL(KTextVeryLongLabel);
    // Edit 2: Uncomment the line below
    User::Leave(KErrNoMemory); // simulates the line below leaving
    iBottomLabel->SetTextL(KTextExtremelyLongLabel);
    );
```

2. After the Trap harness, test the value of the `error` variable. If it is not equal to `KErrNone` reset the text of the top label to "Label", and propagate the leave up the call stack. Enter the following code to do this:

```
if (error != KErrNone)
    {
    iTopLabel->SetTextL(KTextLabel);
    User::Leave(error);
    }
```

3. Save the changes made to the file.
4. Rebuild the application, launch the emulator and navigate to and open the S60MemoryLab application.
5. Once the application is open, select the Options softkey to bring up the options menu.
6. As in the previous task, select the “Toggle Labels” menu item. The information note is displayed again as shown in Figure 9.
7. When the note disappears the screen will have been reset to its original state, as shown in Figure 2.
8. Exit the application.
9. Close the emulator.

Exercise 3 - Using the Cleanup Stack to prevent Memory Leaks

Task 3.1 – Running the application with no memory leaks

This task illustrates what should happen when a string buffer is allocated on the heap and used to set the value of the bottom label of the application.

1. Launch the emulator and navigate to and open the S60MemoryLab application.
2. When the application is open, select the Options softkey to bring up the options menu.
3. Select the “Cleanup Stack Test” menu item from the options menu. The bottom label of the application is changed as shown in Figure 11.
4. Exit the application.
5. Close the emulator.

Figure 11 Screen shot of emulator window after the Cleanup Stack Test menu option has been selected

Task 3.2 – Running the application with a memory leak

Now we shall see what happens when a leave occurs during this operation.

1. In the text editor window showing the contents of the `S60MemoryLabContainer.cpp` file, navigate to the `CS60MemoryLabContainer::CleanupStackTestL()` function and uncomment the line `User::Leave(KErrNoMemory);`. This mimics the effect of the call to `SetTextL()` on the bottom label Leaving.
2. Save the changes made to the file.
3. Rebuild the application, launch the emulator and navigate to and open the S60MemoryLab application.
4. When the application is open, select the Options softkey to bring up the options menu.
5. Select the "Cleanup Stack Test" menu item from the options menu again. The information note is displayed again as shown in Figure 9.
6. Now exit the application, but not the emulator, by pressing the Back key. An error dialog, as shown in Figure 12, is displayed when the application exits. This error dialog indicates that a memory leak has occurred somewhere in the application.

Figure 12 Screen shot of the emulator showing an error dialog indicating a memory leak

The memory leak occurs because the string buffer `(HBufC*)` allocated on the heap in the function `CS60MemoryLabContainer::CleanupStackTestL()` has not been deleted (because of the leave). The emulator checks if all the memory allocated within an application is deleted on exit and if not displays an error dialog.

7. Click OK to close the error dialog.
8. Close the emulator.

Task 3.3 – Preventing memory leaks

The final step illustrates how to prevent memory leaks by using the cleanup stack.

1. In the text editor window showing the contents of the `S60MemoryLabContainer.cpp` file, navigate to the `CS60MemoryLabContainer::CleanupStackTestL()` function and put `labelText` on the cleanup stack just after it is created. This is done using `CleanupStack::PushL(labelText)`. In the event of a leave, all items on the cleanup stack are deleted.
2. In the same function, replace `delete labelText` with `CleanupStack::PopAndDestroy()`. This removes the item from the cleanup stack in addition to deleting it.
6. Save the changes made to the file.
7. Rebuild the application, launch the emulator and navigate to and open the S60MemoryLab application.
3. When the application is open, select the Options softkey to bring up the options menu.
4. Select the “Cleanup Stack Test” menu item from the options menu again. The information note is displayed again as shown in Figure 9.
5. Exit the application without the closing the emulator. There should be no memory leak this time.
6. Close the emulator.
7. Close the Carbide.c++ IDE. This completes the lab.

Developer
Training

Module #04305

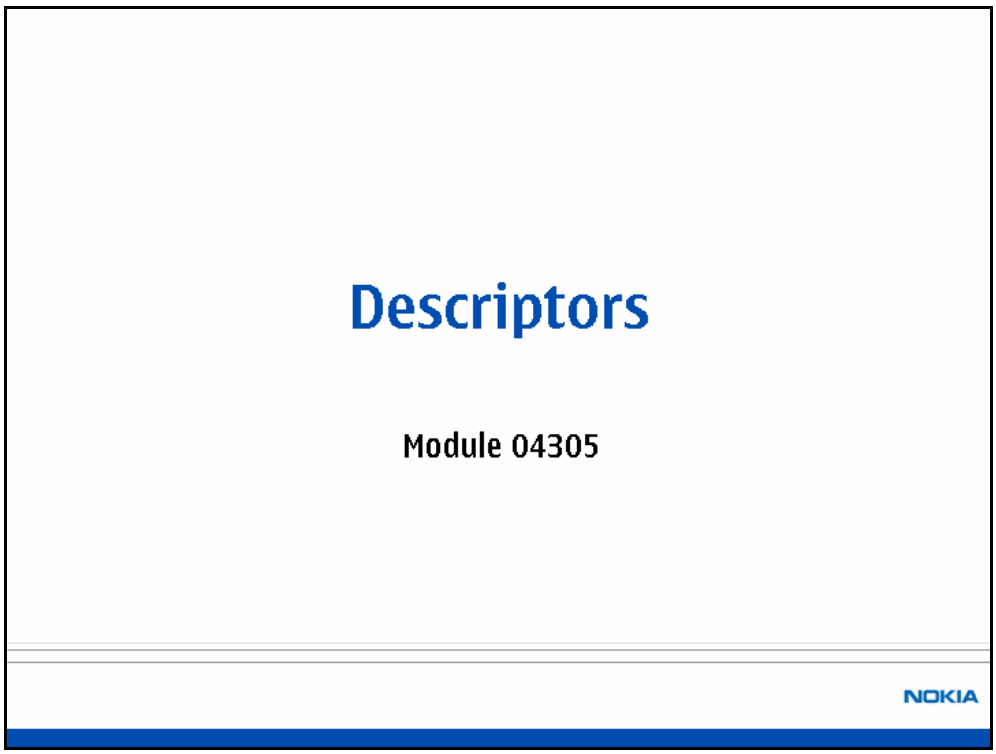
Descriptors

Contents

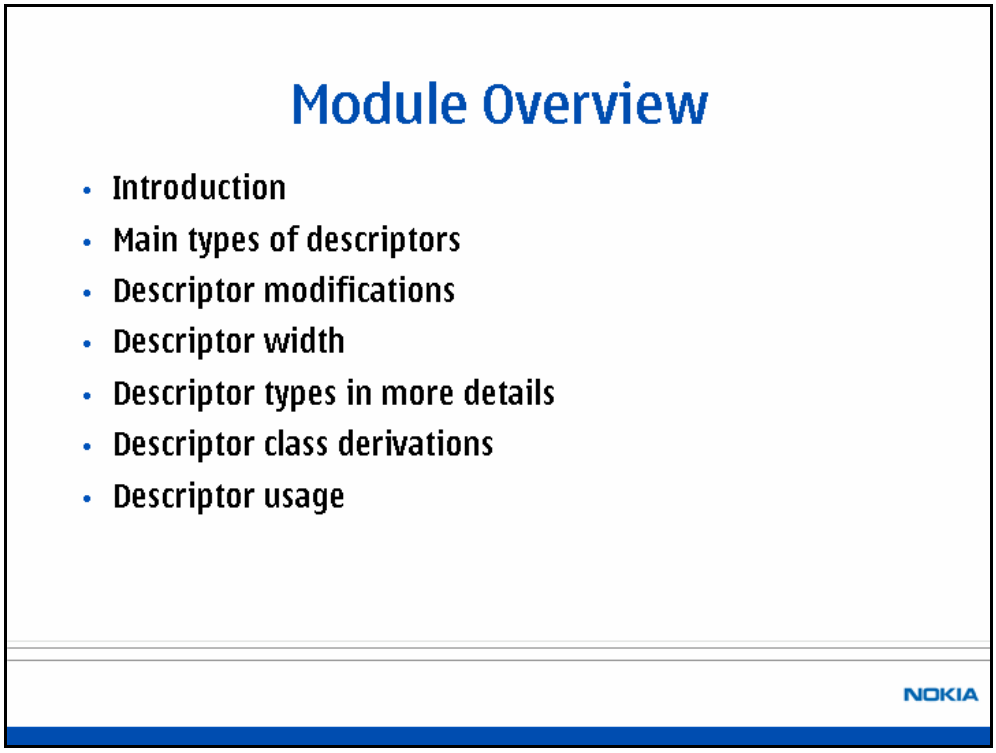
NOKIA

BO-EN-04300-20080108

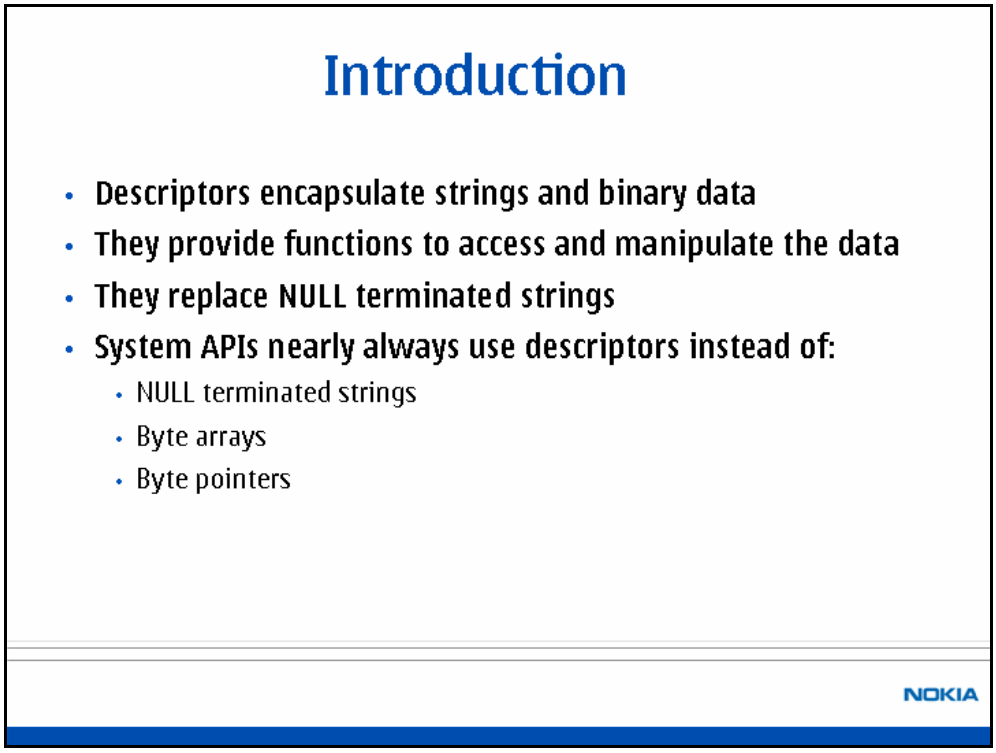
Descriptors



Module Overview



Introduction



Descriptors are a family of classes that are used in Symbian OS for string handling. They are used in preference to NULL-terminated C strings, in all Symbian OS APIs. The advantage of doing this is that the length of the data is encapsulated in the descriptor unlike the NULL-terminated string which requires a `strlen()` to determine its length. Additionally, `char*` should not be used since this does not allow for Unicode strings and is therefore more inflexible to localise.

It is still possible to use a `TText` array to represent a Unicode string:

```
const TText KHelloWorld[] = {'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o',
'r', 'l', 'd'};
```

Note that the following code will not compile since a S60 build is always Unicode:

```
const TText* KHelloWorld = "Hello World";
```

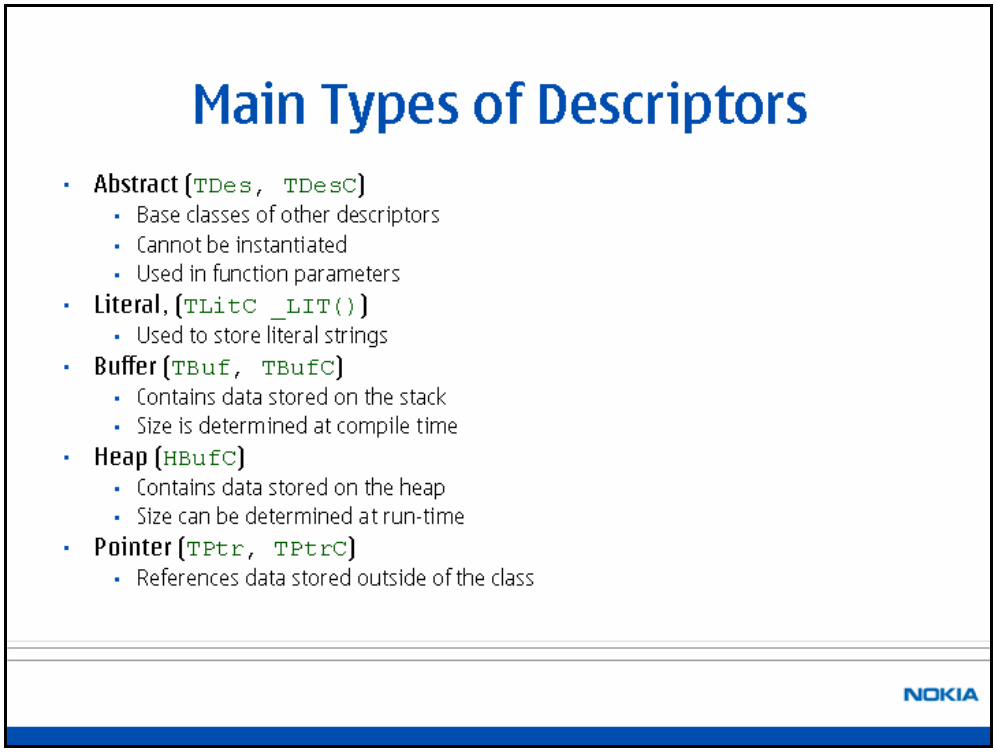
However to display anything to the screen, or to print debug output from a GUI application, we would have to convert it to a descriptor anyway:

```
TPtrC ptr(KHelloWorld);
CEikonEnv::Static()->InfoMsg(ptr);
```

In the code above, the `TPtrC` references the read-only data. It would have been better to store the string in a descriptor in the first place.

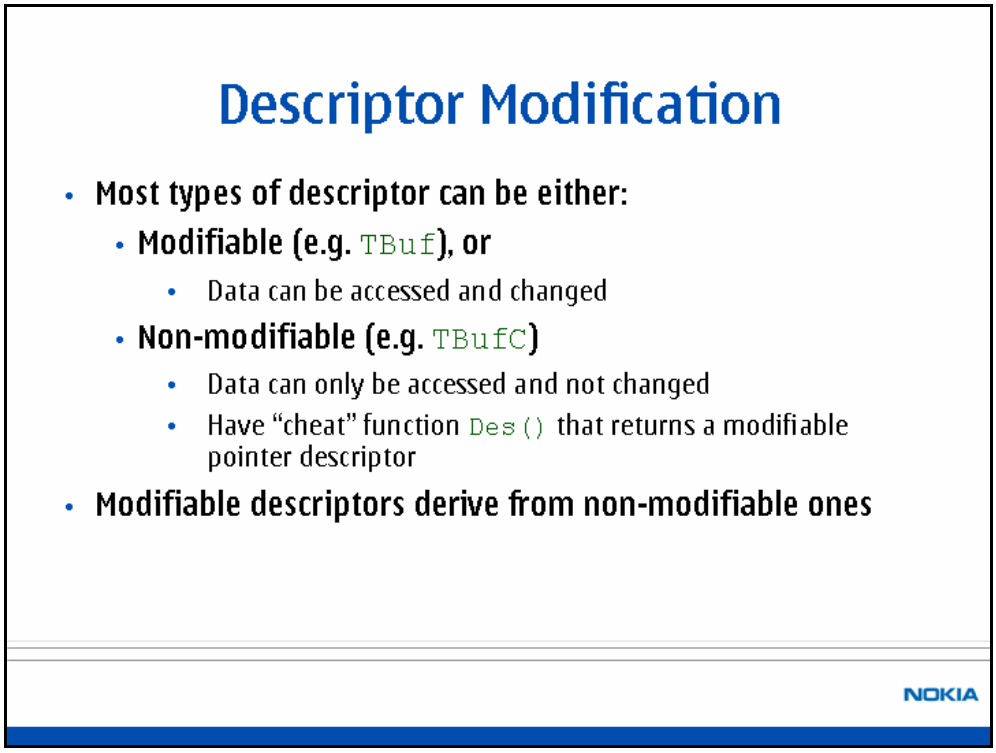
Descriptors can be used for general binary data. Again, APIs that take pointers to binary data require descriptors, so it is not mandatory to use them for this purpose – it just makes life easier.

Main Types of Descriptors



The slide shows the main descriptor types and their associated width-independent classes. Note that we will cover the topic of the width of the descriptor data in a short while. The trailing 'C' on the class name indicates that the descriptor is non-modifiable. Again this will be covered in the next slide. Each of the descriptor types will be covered in turn in greater depth later with usage examples.

Descriptor Modification

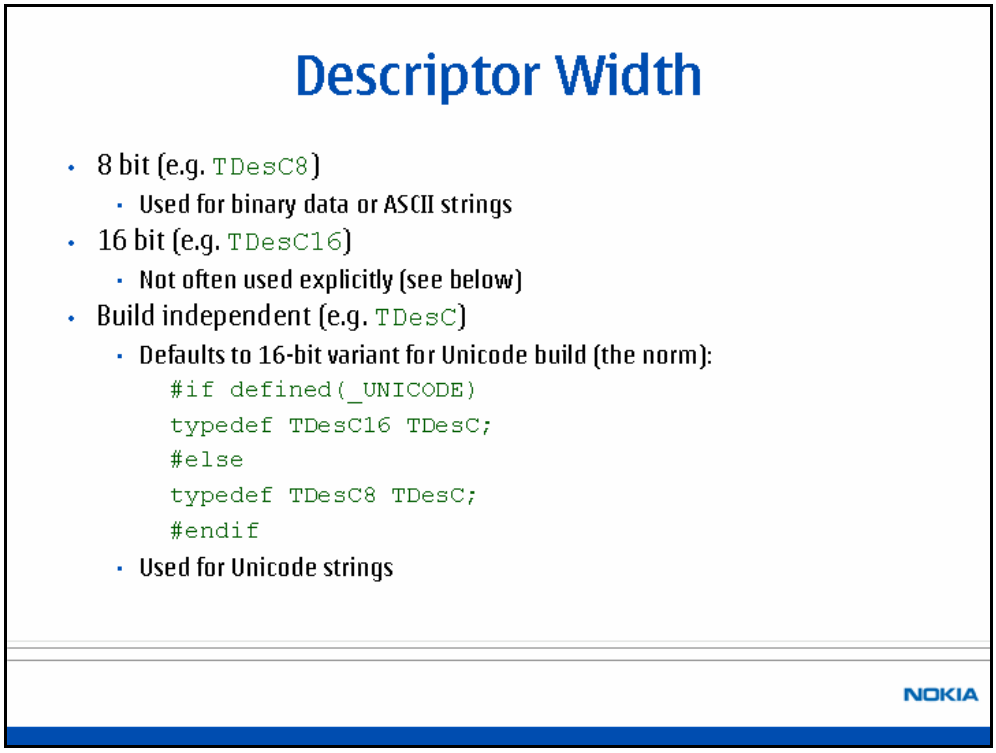


Descriptors can be modifiable and non-modifiable. A 'C' is appended to the basic name of a modifiable descriptor to get the equivalent non-modifiable name.

- Modifiable descriptor classes provide APIs that allow access and modification of data contents, such as appending or replacing characters.
- Non-modifiable descriptor types can have their contents accessed (read-only) but not modified although `HBufC`, `TBufC` and `TPtrC` provide `Set` and `Reset` functionality. There is, however, a cheat function called `Des()` that can be called on classes that returns a modifiable pointer descriptor that references the same data. Obviously this will not work if the data is in ROM (or acting as ROM)!

Non-modifiable descriptors are a base class of modifiable ones, providing the set / reset and read-only access functionality. The modifiable classes extend this functionality to allow data modification.

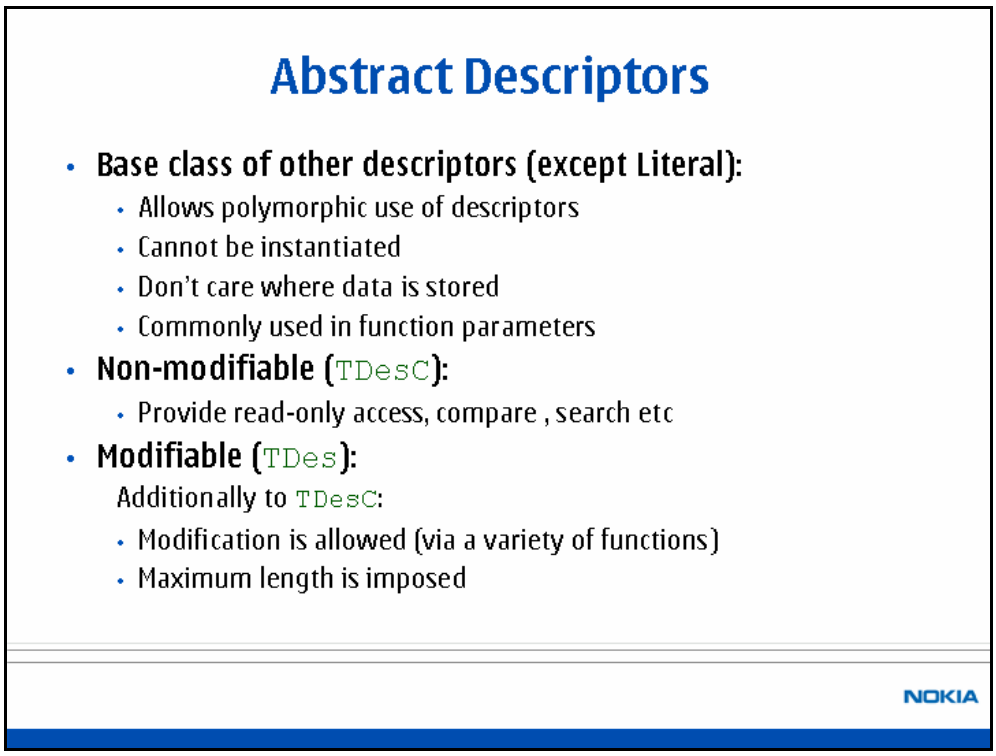
Descriptor Width



Appending the number 8 or 16 to the end of the descriptor class affects the width of the data stored within it or referenced from it. If the width is not specified, the descriptor will be a 16-bit one as only Unicode builds are supported for S60.

The common practice is to use explicit 8-bit descriptors for binary data and build independent descriptors for Unicode strings.

Abstract Descriptors



These descriptors provide the base interface and provide basic functionality for all the descriptors that derive from them. They specify whether the data associated with the descriptor can be modified but not how the data is stored. All the descriptors that are derived from them use the data access and manipulation functions provided by these abstract descriptors. `TDes` and `TDesC` are the abstract descriptors.

Because these classes are abstract, they cannot be instantiated directly. They are typically used in function arguments. Examples of this are given below.

TDesC

`TDesC` is the base class descriptor and provides functions to compare, copy, search, extract portions of the data and access the data directly (but not modify it!). The length of the data in the descriptor is stored and can be returned to the caller. An example use would be to access constant data via a function parameter:

```
TInt TDesCUtil::SumLengths(const TDesC& aDesC1,
                           const TDesC& aDesC2)
    {
    return aDesC1.Length() + aDesC2.Length();
    }
```

The very simple example above just illustrates that to get a string length, we do not need write access to the descriptor nor do we need to know where the string data is stored.

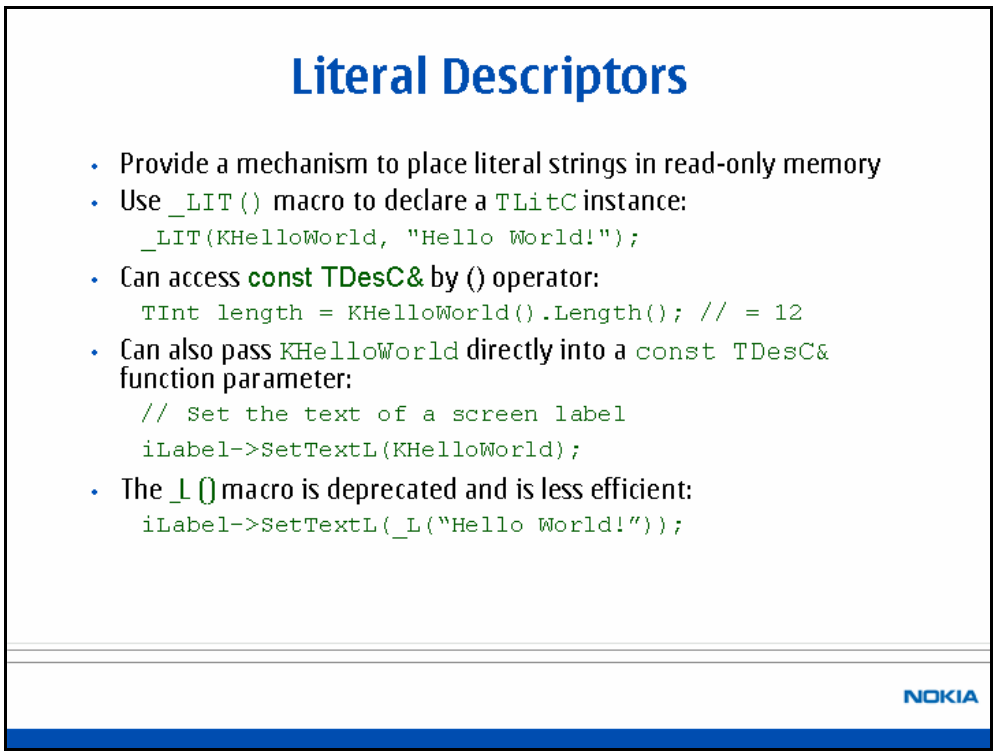
TDes

`TDes` is derived from `TDesC`, and provides many additional functions to modify the data. The concept of a maximum size of data is introduced and is set when the descriptor is created. If attempts are made to modify the data beyond this size, a panic will occur. The example below illustrates an enhanced append function that raises a leave if the data to

be appended to the target would cause a buffer overflow. Normally this would cause a panic which would exit the application.

```
void TDesUtil::AppendL(TDes& aTarget, const TDesC& aDesC)
    {
    TInt sumLengths = aTarget.Length() + aDesC.Length();
    if (aTarget.MaxLength() < sumLengths)
        {
        User::Leave(KErrOverflow);
        }
    aTarget.AppendL(aDesC);
    }
```

Literal Descriptors

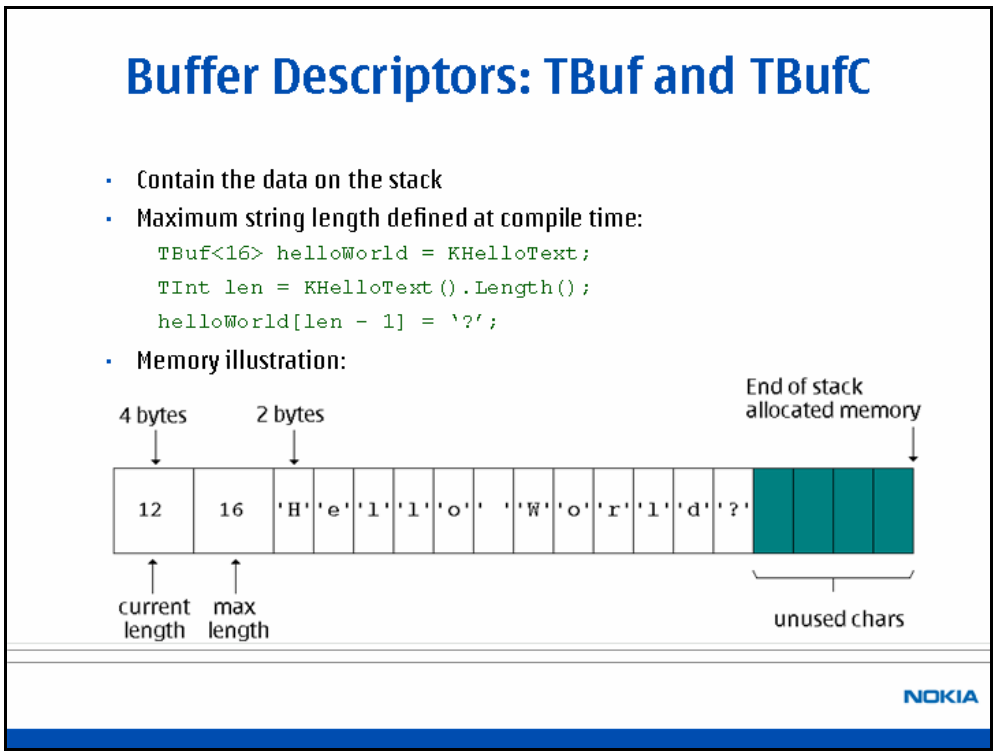


`TLitC` is not commonly used. Instead the `_LIT()` macro is used to specify literal strings. It is also possible to use the `_LIT8()` macro for ASCII strings.

When `_LIT()` and `_LIT8()` macros are used, the data is actually stored within the application/program binaries which are loaded up into RAM. They are not actually stored in ROM (or flash memory behaving as ROM), but can be thought of as read-only.

For applications that require localisation, literal strings should not be used to display text to the user. Instead, strings should be defined in an application's resource and localisation files. These are covered in a later module.

Buffer Descriptors: TBuf and TBufC



These descriptors store their data as part of themselves. They can be declared on the stack (as local or member variables) since they use a fixed amount of memory that is determined at compile time. Their maximum length is set by instantiation of a template. To illustrate the template mechanism, here's part of the `TBuf` declaration:

```
template <TInt S> class TBuf
    {
    …
    };
```

The template is used as follows:

```
TBuf<5> hello; // the size is specified at declaration
```

If the contents of the buffer overflow beyond the maximum length, a USER 11 panic is raised.

TBufC

`TBufC` is a subclass of `TDesC`. It is possible to copy new data into the descriptor to reset the data content. Additionally `TBufC::Des()` can be used to return a modifiable pointer descriptor for the data. This is a way to modify the contents of a TBufC. The maximum length of the pointer descriptor is set to the compile time length of the buffer and cannot be exceeded or else a panic will occur. An example of how a `TBufC` may be used follows. Note the use of `_LIT` to define a literal string (much easier than the method shown in the descriptor introduction!).

```
_LIT(KHelloWorld, "Hello World");
const TInt maxBuf = 32;
…
TBufC<maxBuf> buf; // empty buffer of length 0
```

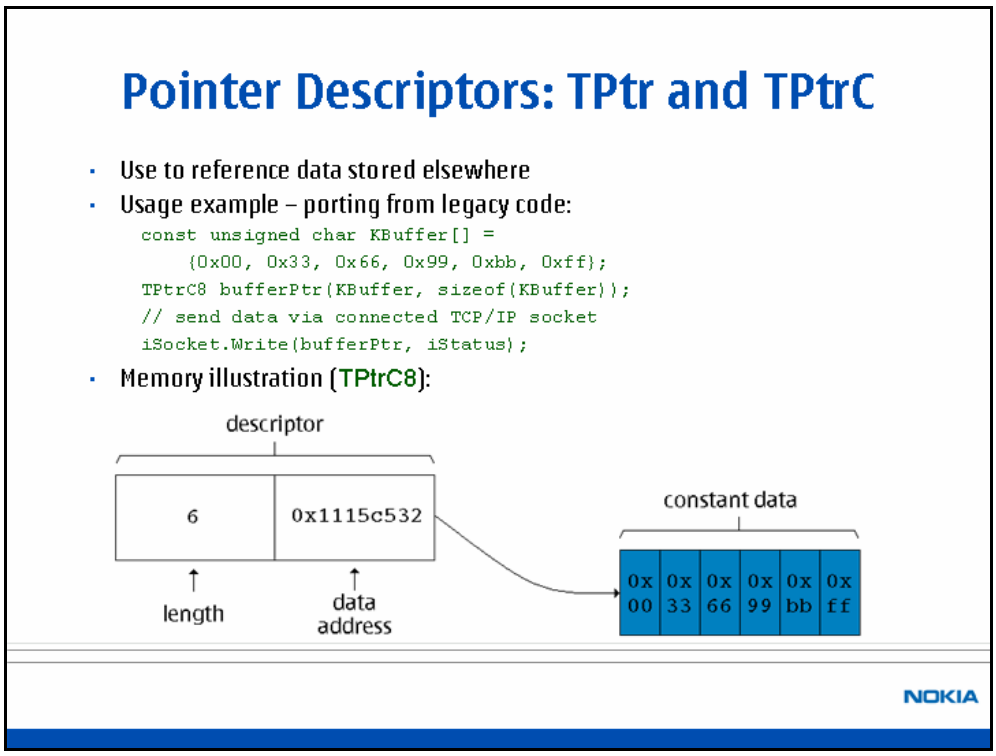
```
TInt currentLen = buf.Length(); // == 0
buf = KHelloWorld; // set the contents after construction
currentLen = buf.Length(); // == 11
TText ch = buf[2]; // == 'l'
```

TBuf

`TBuf` is a subclass of `TDes`. In addition to providing many functions to modify the data (in `TDes`), new data can be copied into the buffer as long as the maximum length is not exceeded. The maximum length is set to the integer value with which the template was instantiated.

```
const TInt bufLen = 6;
TUint8 objType = 1;
TUint8 objId = 1;
TUint8 xCoord = 128;
TUint8 yCoord = 192;
TUint8 xVelocity = 5;
TUint8 yVelocity = 0;
TBuf8<bufLen> buf;
buf.Append(objType);// type of object to store
buf.Append(objId);// id of the object to store
buf.Append(xCoord);// x-coordinate of object to store
buf.Append(yCoord);// y-coordinate of object to store
buf.Append(xVelocity);// x-velocity of object to store
buf.Append(yVelocity);// y-velocity of object to store
// we can now do something with the buffer
// such as write it to a binary file…
```

Pointer Descriptors: TPtr and TPtrC



Refers to data stored elsewhere that is not owned by the descriptor. Using a `TPtr` or `TPtrC` to access a string is much safer than maintaining pointers to zero-terminated C strings.

TPtrC

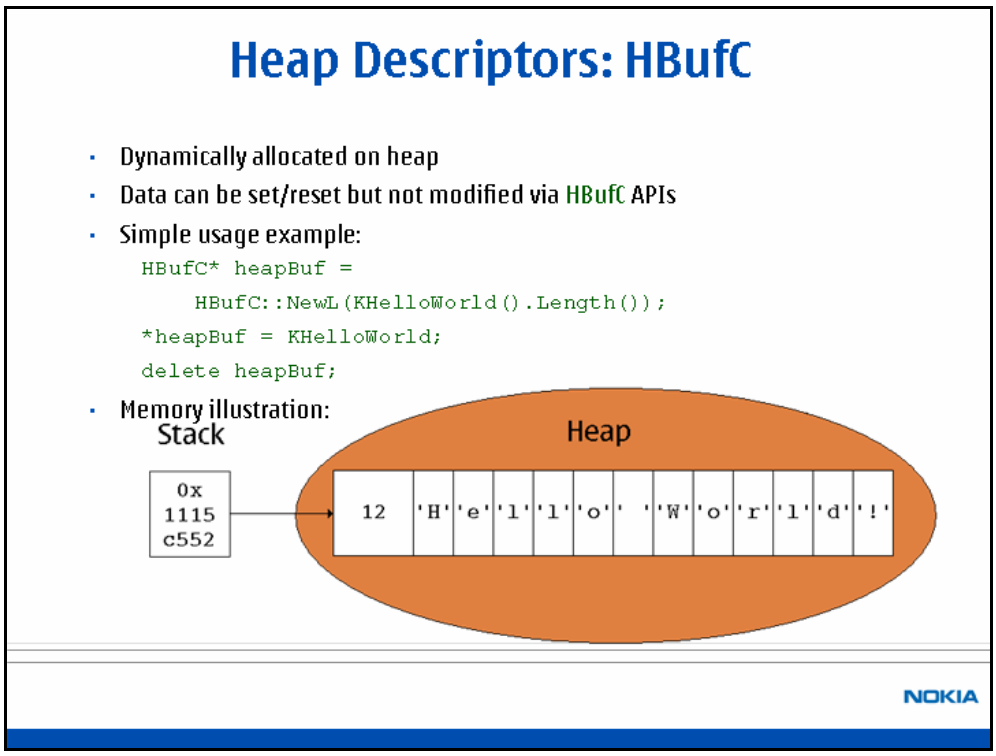
`TPtrC` is derived from `TDesC` and only provides additional methods to associate the pointer with the constant data buffer. No methods to reset or modify the data are provided. See the descriptors overview section for example usage of a `TPtrC`.

TPtr

`TPtr` is derived from `TDes` and provides additional methods to associate the pointer with the data buffer, set the maximum length, and copy new data into the referenced buffer (overwriting the old data). Here is an example:

```
_LIT(KHelloWorld, "Hello World");
TBufC<maxBuf> buf;
buf = KHelloWorld; // set the contents
TPtr ptr = buf.Des(); // obtain a pointer to the buffer
ptr[7] = 'a'; // Change the 'o' to an 'a'
ptr[8] = 'l'; // Change the 'r' to an 'l'
ptr[9] = 'e'; // Change the 'l' to an 'e'
ptr[10] = 's'; // Change the 'd' to an 's'
CEikonEnv::Static()->InfoMsg(ptr);
// The buffer now holds "Hello Wales"
```

Heap Descriptors: HBufC

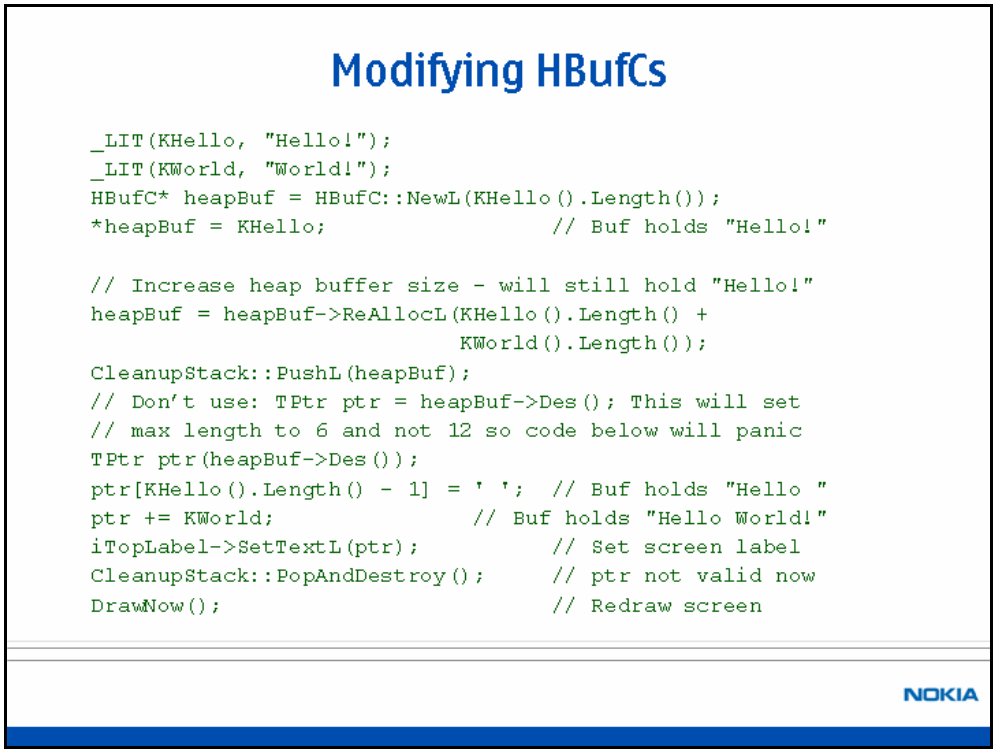


This encapsulates data stored on the heap that is owned by the descriptor. The data is allocated dynamically and so the maximum length of this descriptor can be set and changed at run-time by reallocating the heap buffer. `HBufC` is based on `TDesC` and provides methods to change the size of the data stored (HBufC::ReAlloc() and `HBufC::ReAllocL()`) and an assignment operator to set the contents of the data (subject to the maximum length not being exceeded). The contents can be modified by using `HBufC::Des()` to obtain a modifiable pointer descriptor.

`HBufC`s are commonly used in the scenarios where the length of a returned string does not need to be known by the caller:

- Loading strings from a resource file at run-time.
- Getting strings from UI elements, for example query dialogs.
- Getting strings from application engines, for example names from the contacts database.

Modifying HBufCs



The example on the slide illustrates how `HBufC`s can be modified and also expanded in size. A `HBufC` is initially created to be 6 (Unicode) characters long and to contain the string `"Hello!"`. Remember that `HBufC`s can have their data set and reset, but not modified directly by the `HBufC` APIs.

The next section of the code reallocates the heap buffer so that it is 12 Unicode characters long. The `ReAllocL()` function copies the existing `"Hello!"` data into the newly reallocated buffer and sets the length to 6. It also deletes the previous heap buffer. The heap buffer is then put on the cleanup stack so that if the following code leaves there will not be a memory leak.

`HBufC::Des()` is used to get a pointer descriptor which is used to modify the contents of the heap buffer. Note that the `TPtr(const TPtr& aTPtr)` constructor overload is used to pass in the `TPtr` returned from `HBufC::Des()`. This will copy the contents of the heap buffer into the pointer descriptor, set the length to 6 and the max length to 12. By contrast, if `TPtr ptr = heapBuf->Des();` were used, the max length of the newly declared `TPtr` would be 6. This would cause the `ptr+= KWorld;` statement to panic since the max length would be exceeded. If the heap buffer was reallocated in the future, the pointer descriptor should be updated by calling:

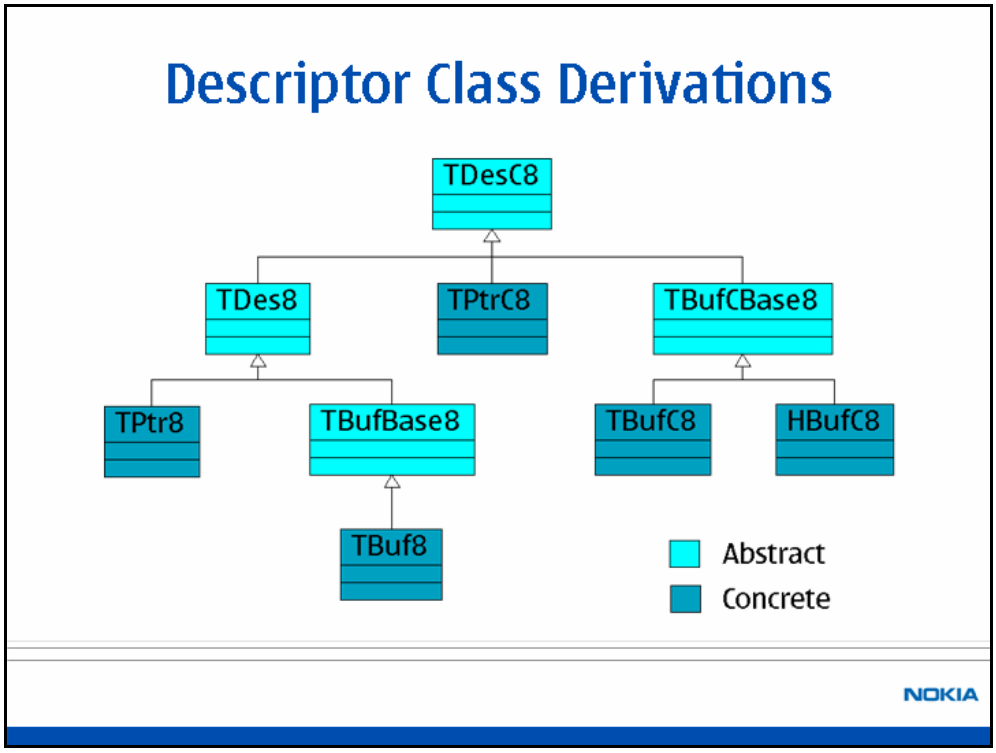
```
ptr.Set(heapBuf->Des());
```

Once the pointer descriptor is constructed on the slide, it can be used to modify the heap buffer string. Individual characters can be modified - `'!'` is changed to a space above. The slide also shows an example of a string being appended onto the current string.

The contents of the heap buffer are displayed on the screen (this assumes the program is a GUI application). The function `CEikLabel::SetTextL(const TDesC& aText)` is called to update the text of a label that is stored in a member variable. The `TPtr` passed in is implicitly cast down to a `TDesC` (a base class) which is the parameter type expected.

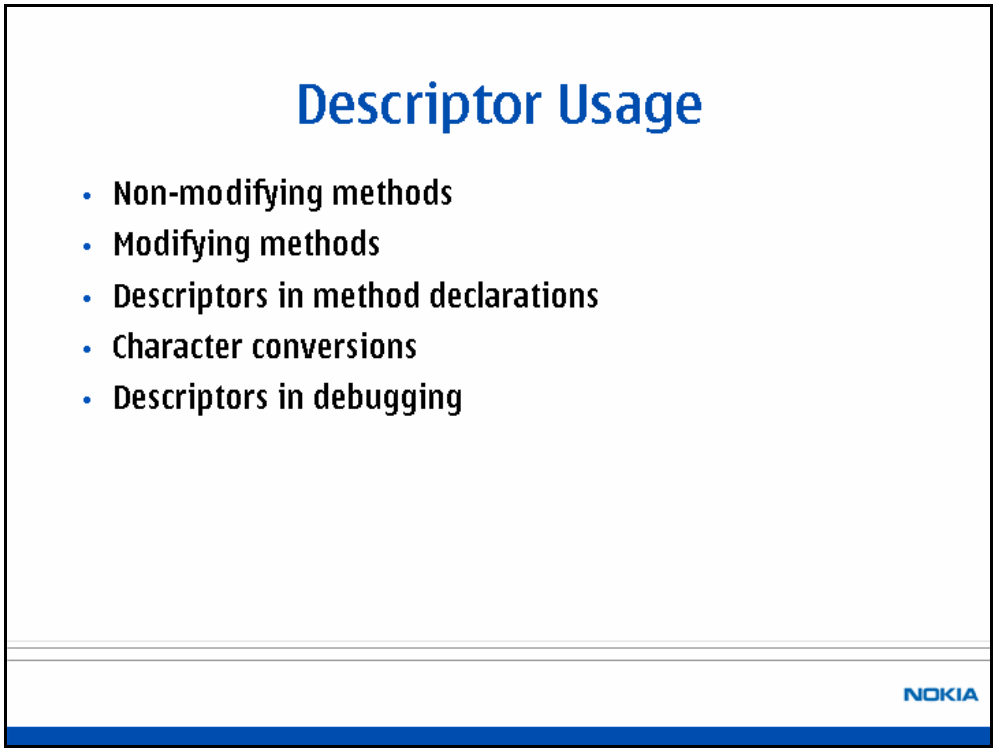
Finally the heap buffer is deleted by calling `CleanupStack::PopAndDestroy()`. After this is called, the pointer descriptor's data is no longer valid (since it still points to the now deleted heap buffer data). The call to `CCoeControl::DrawNow()` requests that the screen is updated.

Descriptor Class Derivations

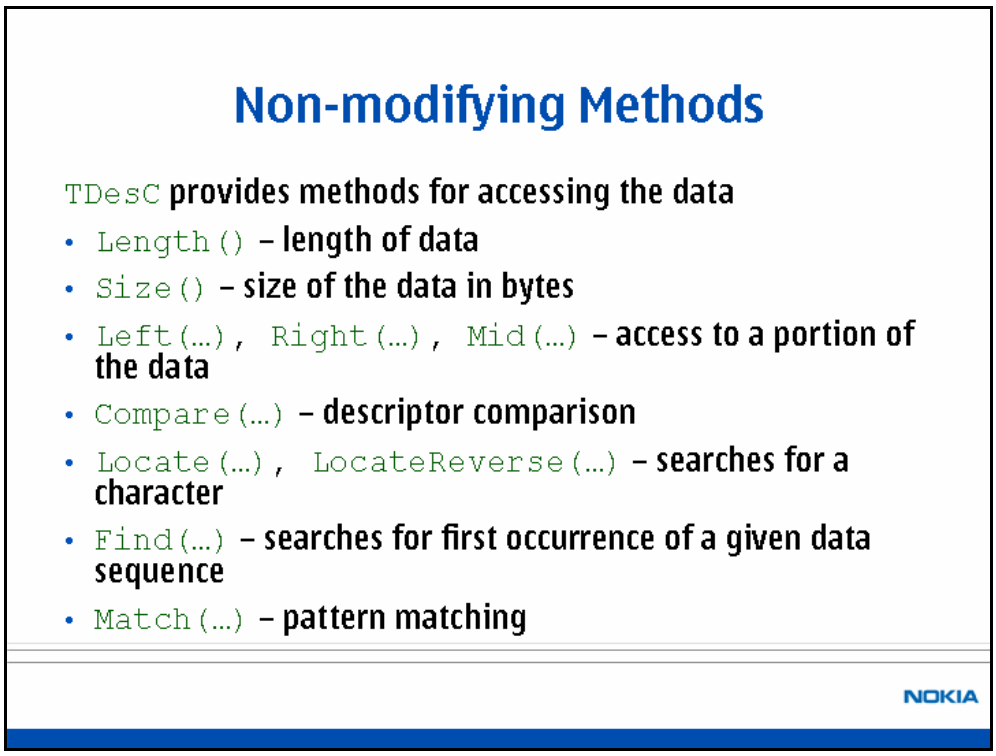


The slide shows the class hierarchy for 8-bit descriptors. The 16-bit descriptor class hierarchy is exactly the same as this except that any number '8' is replaced with '16'. The hierarchy shows that all descriptors are derived from `TDesC8`, so whenever a descriptor pointer of any type is required a `TDesC8` pointer can be used instead. The classes `TBufBase8` and `TBufCBase8` do not provide any public functions (the same as their 16-bit equivalents) and are not normally used.

Descriptor Usage



Non-modifying Methods



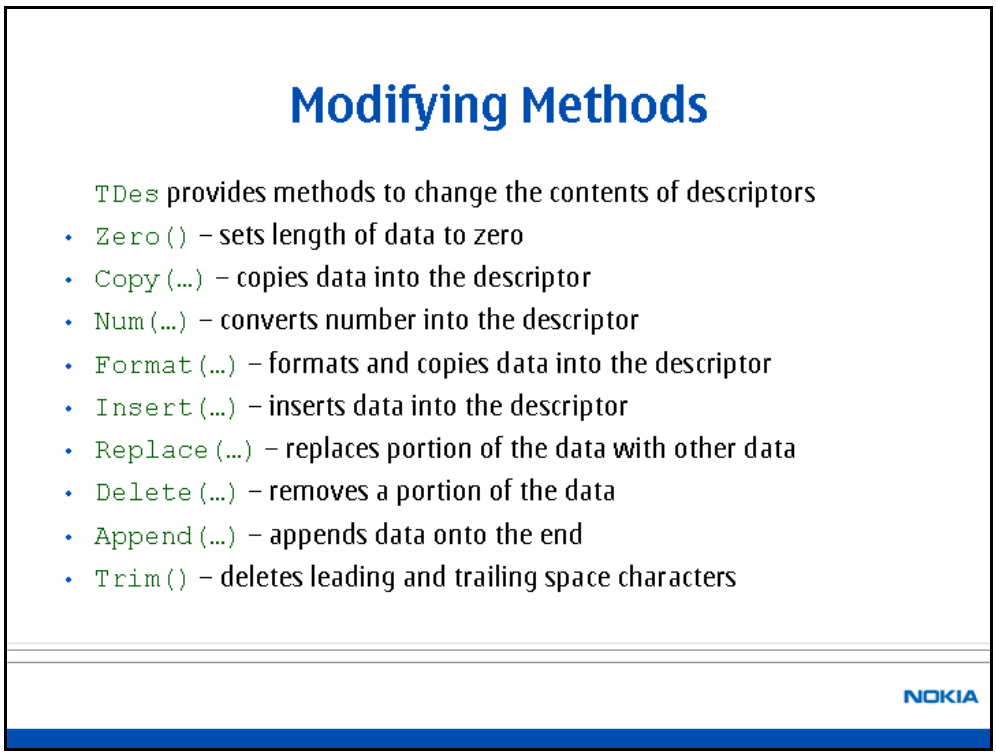
All descriptors are derived from `TDesC`. It has methods for accessing the data but doesn't provide any methods for changing the data. Here is a list of some of these methods.

- `Length()` – Returns the length of the data.
- `Size()` – Returns the size of the data in bytes.
- `Left(…)` – Returns a non-modifiable pointer descriptor to represent the leftmost part of the data.
- `Right(…)` – Returns a non-modifiable pointer descriptor to represent the rightmost part of the data.
- `Mid(…)` – Returns a non-modifiable pointer descriptor to represent a portion of the data.
- `Compare(…)` – Compares whether a second descriptor is lexically more than, less than, or equal to the current instance.
- `Locate(…)` – Returns offset of the character position from the beginning of the data.
- `LocateReverse(…)` – Returns offset of the character position from the beginning of the data starting from the end of the data.
- `Find(…)` – Searches for the first occurrence of the specified data.
- `Match(…)` – Pattern matches data allowing wild characters `'*'` and `'?'`.

A simple example to access data in descriptors is the following code snippet that will return a portion of a given string that is located between the '<' and '>' characters if present or will return the whole string if those characters are not present.

```
static const TUint KAddressStartChar ='<';
static const TUint KAddressEndChar = '>';

TPtrC ExtractAddressNumber(const TDesC& aAddressString)
    {
    TInt addrStart = aAddressString.Locate(KAddressStartChar) + 1;
    TInt addrEnd = aAddressString.LocateReverse(KAddressEndChar);

    if ((addrStart == KErrNotFound) ||
        (addrEnd == KErrNotFound) ||
        (addrStart >= addrEnd))
        {
        addrStart = 0;
        addrEnd = aAddressString.Length();
        }

    return (aAddressString.Mid(addrStart, (addrEnd - addrStart)));
    }
```

Modifying Methods



`TDes` is a base class for modifiable descriptors. Here is a list of methods that change the contents of the stored data:

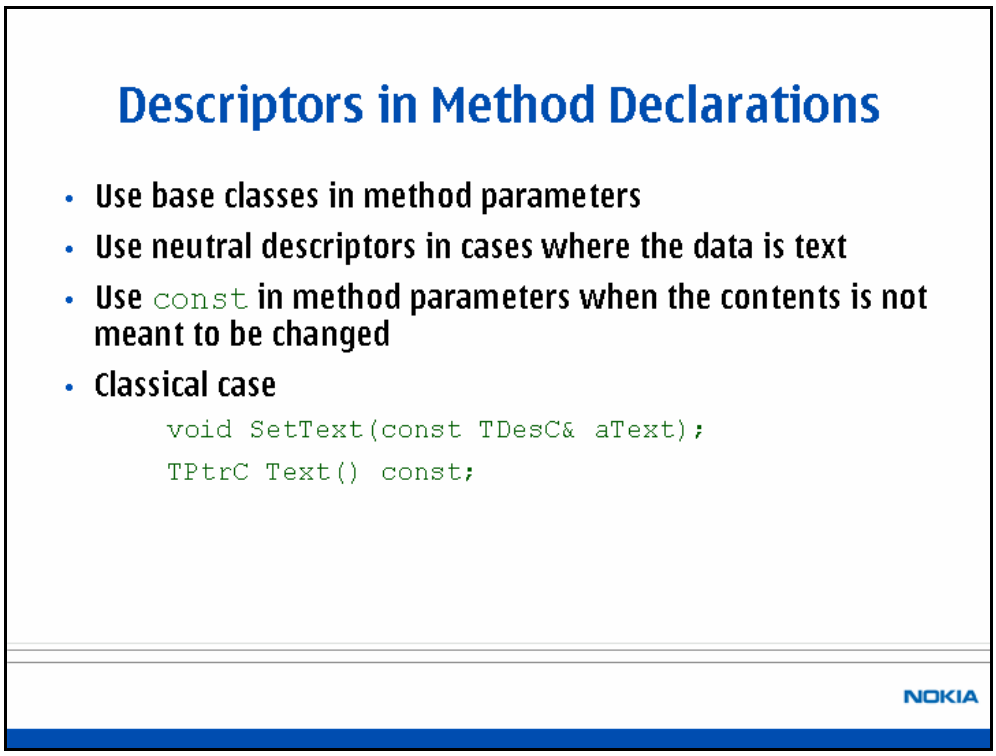
- `Zero()` – Sets the length of the data to zero (a way to clear a descriptor).
- `Copy(...)` – Copies data into descriptor replacing any existing data.
- `Num(...)` – Converts a number into a character representation and stores it into the descriptor.
- `Format(...)` – Formats and copies text into this descriptor, replacing any existing data using formatting strings.
- `Insert(...)` – Inserts data into a descriptor.
- `Replace(...)` – Replaces a portion of the data with other data.
- `Delete(...)` – Removes a portion of data from the descriptor.
- `Append(...)` – Appends data onto the end of descriptor's data.
- `Trim(...)` – Deletes leading and trailing space characters from the descriptor's data.

Below is an example of how to modify data in descriptors:

```
_LIT(KText,"Hello World!");
_LIT(KNewText,"New Text");
_LIT(KReplaced,"Replaced");
…
TBuf<16> buf1(KText);
buf1.Delete(6,6); // length is now 6, leaving "Hello" in the buffer
```

```
...
TBuf<16> buf2(KNewText);
...
buf2.Copy(KReplaced); // buf2 now contains “Replaced”
buf2.Append(KNewText); // buf2 now contains “Replaced New Text”
buf2.Delete(99,1); // attempt to delete data outside of the boundary
// will cause panic
```

Descriptors in Method Declarations



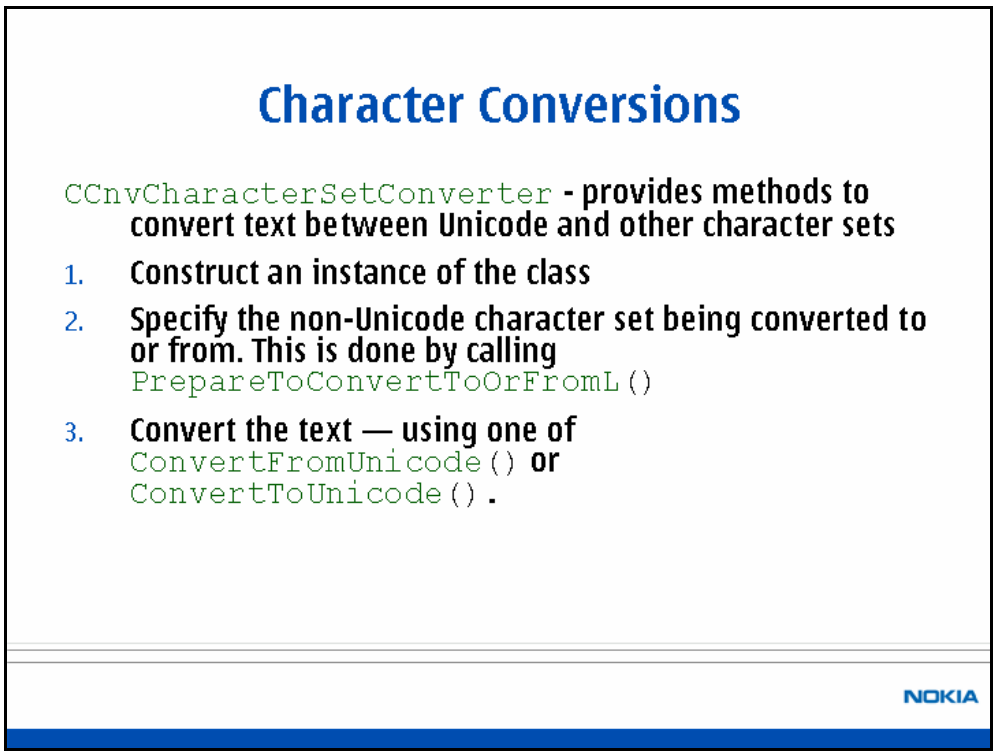
The following best practice should be kept in mind when declaring methods:

- Use base classes in method parameters. If the method does not need any specific properties of a derived class, it is better to use a base class instead of a derived one. It makes it possible to pass all derived classes for the method instead of a specific derived one. So, use `TDesC` or `TDes` in parameter declarations.
- Use neutral descriptors in cases where data is text. When the method parameter is declared to take an 8-bit descriptor, it is assumed to contain binary data. On the other hand, a 16-bit descriptor in the method declaration indicates that the method is specifically handling Unicode data. So, in general, use `TDesC` and not `TDesC8` or `TDesC16`.
- Use const when the contents of a descriptor parameter are not meant to be changed.

For example, the classical methods for setting and getting text should look like this:

```
void SetText(const TDesC& aText);
TPtrC Text() const;
```

Character Conversions



When descriptors are used as strings, their array consists of numbers that represent characters. Each number means a specific character in a character set. Many existing services, protocols and file formats use 8-bit character width and some specific encoding which is different from the Unicode one. Symbian OS has the `CCnvCharacterSetConverter` class, which provides methods to convert text between Unicode and other character encodings. In fact, if conversion is needed between two non-Unicode encodings, the original can first be converted to Unicode encoding and then to the second one.

The usage of `CCnvCharacterSetConverter` is as follows. First, an instance of the class is constructed. Then, the needed non-Unicode encoding ID is given to that instance. Finally, the data is converted from non-Unicode to Unicode or vice versa.

You do not need to use `CCnvCharacterSetConverter` for ASCII-encoded strings since it is a subset of Unicode. The example below converts an ASCII-encoded string to Unicode:

```
TBuf16<64> UnicodeBuf;
_LIT8(KAsciiStr, "Hello");
UnicodeBuf.Copy(KAsciiStr);
```

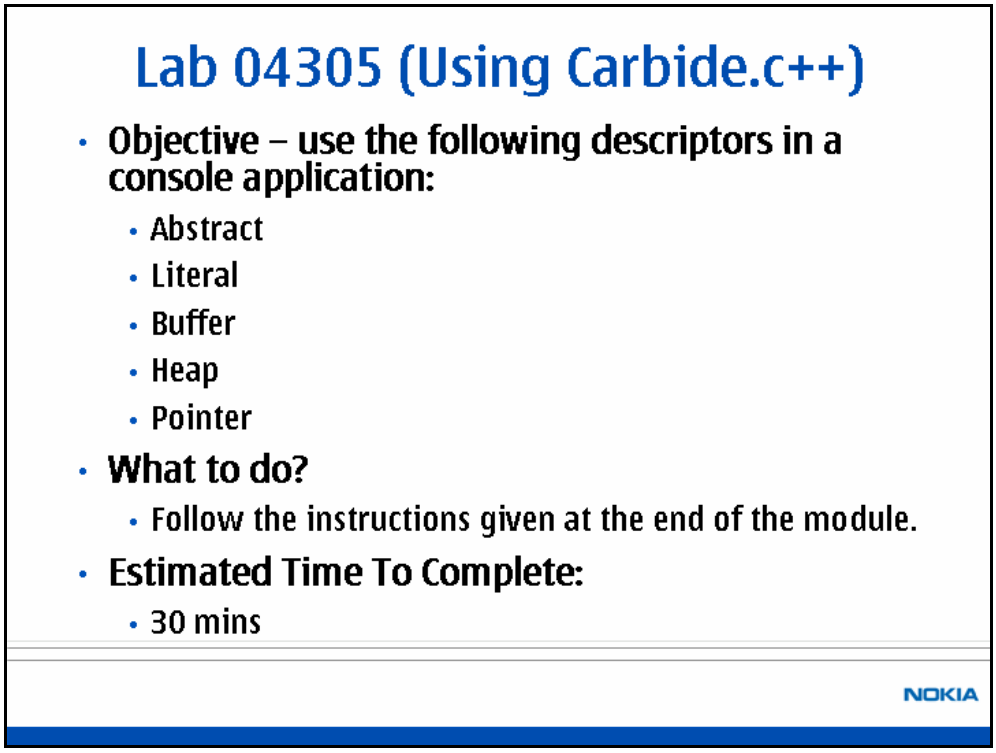
To convert Unicode string to Latin1 encoding do the following:

```
TBuf8<64> Latin1Buf;
_LIT16(KUnicodeStr1, "Hello");
_LIT16(KUnicodeStr2, "I have got 10\x20AC."); // \x20AC is a euro
Latin1Buf.Copy(KUnicodeStr1); // ok
Latin1Buf.Copy(KUnicodeStr2); // the euro sign does not belong to
the // Latin1 set, so it will come as the character
```

The following code is an example of using `CCnvCharacterSetConverter` to convert an SMS-encoded string into Unicode:

```
// 1. Create session to the file server
RFs fsSession;
fsSession.Connect();
CleanupClosePushL(fsSession);

// 2. Create character converter that can convert between
// SMS encoding and UCS-2
CCnvCharacterSetConverter *converter;
converter = CCnvCharacterSetConverter::NewLC();
converter->PrepareToConvertToOrFromL(
KCharacterSetIdentifierSms7Bit,fsSession );

// 3. Declare descriptor containing a string encoded with 7-bit SMS
encoding.
const TUint8 SMSSourceBytes[] =
    {
    54, // character '6' (takes one byte)
    2, // character '$' (takes one byte)
    27, 61, // character '~' (takes two bytes)
    53, // character '5' (takes one byte)
    27, 101, // euro character (takes two bytes)
    };
TPtrC8 SMSEncodedStr(SMSSourceBytes,sizeof(SMSSourceBytes));

// 4. Convert the SMS encoded message to the UCS-2 encoding
TInt state = CCnvCharacterSetConverter::KStateDefault;
TBuf<64> UnicodeBuf;
TInt returnValue = converter->ConvertToUnicode(UnicodeBuf,
SMSEncodedStr, state);
// returnValue is either the number of unconverted bytes left at the
// end of the input descriptor (e.g. because the output descriptor
is // not long enough to hold all the text), or an error values if <
0
// so in the general case an examination of returnValue is advisable

// 5. remove objects from the cleanup stack. Variable
// UCS2Buf has now the SMS characters as unicode
// characters
CleanupStack::PopAndDestroy(2); // converter, fsSession
```

Lab 04305 (Using Carbide.c++)



Lab 04305.cb1

Overview

Title:	Using Descriptors		
Overview:	The objective of this lab is to illustrate how to use descriptors in various kinds of situations.		
Objectives:	To understand how to use the following types of descriptor for various scenarios: • Abstract in function parameters. • Literal to define literal Unicode strings. • Buffer to receive a string entered by the user up to a maximum character limit. • Heap to receive a string entered by the user with no maximum character limit. • Pointer to modify a heap buffer and to search a non-modifiable abstract descriptor.		
Compatible IDE(s):	Carbide.c++ v1.2		
Compatible SDK(s):	S60 3rd Edition, S60 3rd Edition (MR), S60 3rd Edition FP1		
App. Type:	Console	**App. Name:**	DescriptorLab.exe
Starter Code Provided:	Yes	**Solution Code Provided:**	Yes
Estimated Time To Complete:	30 minutes		

Lab Instructions

Exercise 1 – Preliminary Steps

Task 1.1 – Checks

1. Ensure that a compatible SDK has been installed. A list of compatible SDKs to use with this lab is given in the table in the Introduction.
2. Ensure that a compatible IDE has been installed. A list of compatible IDEs to use with this lab is given in the table in the Introduction.
3. Ensure that the lab files are located in the folder `C:\Labs\Lab_04305.cb1\`. If this is not the case please refer to the setup guide for details of how to obtain them.

Task 1.2 – Make sure the emulator is in text shell mode

1. Open Windows Explorer.
2. Navigate to the file named epoc.ini, located in the `<EPOCROOT>\epoc32\data` folder. Here, `<EPOCROOT>` is a placeholder for the root folder location of your SDK.

3. Open the epoc.ini file using Notepad.
4. Check that the file contains the “textshell” statement. If the statement is NOT present in the file, then add it to the beginning of the file as shown by the text in bold below:

```
textshell
<Other statements in epoc.ini file>
...
```

5. Press ‘Ctrl’ + ‘S’ to save the contents of the file (if you altered it) and close Notepad.
6. Close Windows Explorer.

Task 1.3 – Carbide Setup

1. Launch the Carbide.c++ IDE. Depending on your Carbide.c++ settings the IDE will either display the “Workspace Launcher” dialog or open the last used workspace. If the latter occurs open the “Workspace Launcher” dialog by selecting the “File -> Switch Workspace...” menu item.
2. In the workspace launcher window, create a new workspace by typing `C:\Labs\<Workspace>` in the Workspace field. Here `<Workspace>` is a placeholder for a folder name of you choosing that does not already exist and does not contain spaces.
3. Click the OK button.
4. Close the Welcome window after it appears when the workspace is opened.

Task 1.4 – Importing the project

15. In the Carbide.c++ IDE, select the “File > Import” menu item. The Import dialog appears.
16. Select “Symbian OS bld.inf file” from the “Symbian OS” subfolder as the import source and click the Next button. A new screen appears on the Import dialog.
17. Enter `C:\Labs\Lab_04305.cb1\starter\group\bld.inf` in the “bld.inf file” field. Alternatively, click the “Browse...” button and then navigate to and select the correct file via the dialog. When this is done a list of configurations appears in the “SDKs and Build Configurations pane”.
18. Expand the list of configurations for your installed SDK and tick the checkbox next to the “Emulator Debug” configuration. (Note that other configurations may be chosen in addition to this one, but they are not used in this lab.) Then select the “Next” button.
19. Ensure the MMP file and all the make files are selected on the “MMP Selection” screen. Then select the Next button. The Project Properties screen appears.
20. Change the name in the Project field to “DescriptorLab” (without the quotes) and click the Finish button to complete the import.
21. The Import dialog closes and a new project called “DescriptorLab” appears in the “C/C++ Projects” view.

Exercise 2 – Using descriptors

In this exercise pay special attention when entering variable names since misspelling them is a common source of compilation errors.

Task 2.1 – Using a buffer descriptor

This task illustrates how to declare and use a buffer descriptor.

1. Double click on the source file, DescriptorLab.cpp, located in the DescriptorLab folder in the "C/C++ Projects" view. A text editor window is opened showing the contents of the file.
2. In the `CDescriptorLab::UseBufferDes()` function, uncomment the line following the "Edit 1" comment to declare a literal string.
3. Under the "Edit 2" comment, declare a `TBuf` named `buf` of max size `KBufSize`. Initialise the length of the buffer to zero. Enter the following code to do this:

   ```
   TBuf<KBufSize> buf(0);
   ```

4. Uncomment the 2 lines of code following the "Edit 3" comment to get a string of up to `KBufSize` characters from the user. The code requests a character from the user and then calculates the number of occurrences of that character in the string `buf`. The result is printed to the screen.

Task 2.2 – Using an abstract modifiable descriptor

This task shows how to use an abstract modifiable descriptor to get a string from the user.

1. In the `CDescriptorLab::GetStringFromUser()` function, under the "Edit 4" comment, declare an integer called `maxLen` that is initialised to the maximum length of the descriptor `aBuf` parameter. Enter the following code to do this:

   ```
   TInt maxLen = aBuf.MaxLength();
   ```

2. Under the "Edit 5" comment, append the character, entered by the user, to the `aBuf` descriptor. Enter the following code to do this:

   ```
   aBuf.Append(key);
   ```

Task 2.3 – Using TPtrC and TDesC

This task illustrates the use of the `TPtrC` and `TDesC` descriptor classes.

1. Locate the `CDescriptorLab::CharOccurance()` function which gets a char from the user and counts the number of occurrences in a supplied string.
2. Under the "Edit 6" comment, declare an object of type `TPtrC` called `subStr`. Initialise it to the `aSearchStr` parameter.
3. Uncomment the line of code below the "Edit 7" comment. This code finds the position of the first occurrence of the char entered by the user in the `aSearchString` parameter.
4. Uncomment the line of code below the "Edit 8" comment to reset `subStr` to point to the remainder of `aSearchStr` after the position of the last found char.

Task 2.4 – Using a heap buffer descriptor

This task shows how to use a heap buffer descriptor.

1. In the `CDescriptorLab::UseHeapDesL()` function, under the "Edit 9" comment, declare an object of type `HBufC*` called `buf` that is initialised to the return value of a call to `CDescriptorLab::StringFromUserL()`.
2. Under the "Edit 10" comment, call `CDescriptorLab::CharOccurance()` passing the dereferenced value of `buf`.
3. Under the "Edit 11" comment, delete the memory associated with `buf`.

Task 2.5 – Using HBufC and TPtr

This task illustrates the use of the `HBufC` and `TPtr` descriptor classes.

1. Locate the `CDescriptorLab::StringFromUserL()` function which gets a variable length string with no maximum limit from the user.
2. Under the "Edit 12" comment, declare an object of type `HBufC*` called `heapBuf` and construct via `HBufC::NewL(KGranularity)`.
3. Under the "Edit 13" comment, declare a `TPtr` object called `ptr` that is initialised to `heapBuf->Des()`.
4. Uncomment the line below the "Edit 14" comment to increase the maximum length of the `heapBuf` descriptor whilst retaining its existing contents.
5. Under the "Edit 15" comment, reset `ptr` to point to the new location of `heapBuf`. Hint: call `ptr.Set(heapBuf->Des())` rather than `ptr = heapBuf->Des()` as the latter will cause a panic.
6. Save the contents of the file by selecting the "File -> Save" menu item.

Task 2.6 – Building and running the application

1. Build the application for the Emulator Debug configuration of the SDK you are using by right clicking on the DescriptorLab project in "C/C++ Projects" view and select the "Build Project" or "Rebuild Project" menu item. The project should build successfully with no errors.
2. Run the console application by right clicking on the DescriptorLab project in the "C/C++ Projects" view and selecting the "Run As > Run..." menu item. The Run dialog appears.
3. Click the New button on the Run dialog. A "Run configuration settings" window appears. Accept all the default settings apart from the "Emulator or host application" field which should be made empty; i.e. its contents should be removed.
4. Click the Run button to run the console application. The buffer descriptor example is run followed by the heap descriptor example. Follow the instructions on the screen. Example screen shots are shown below in the following figures.

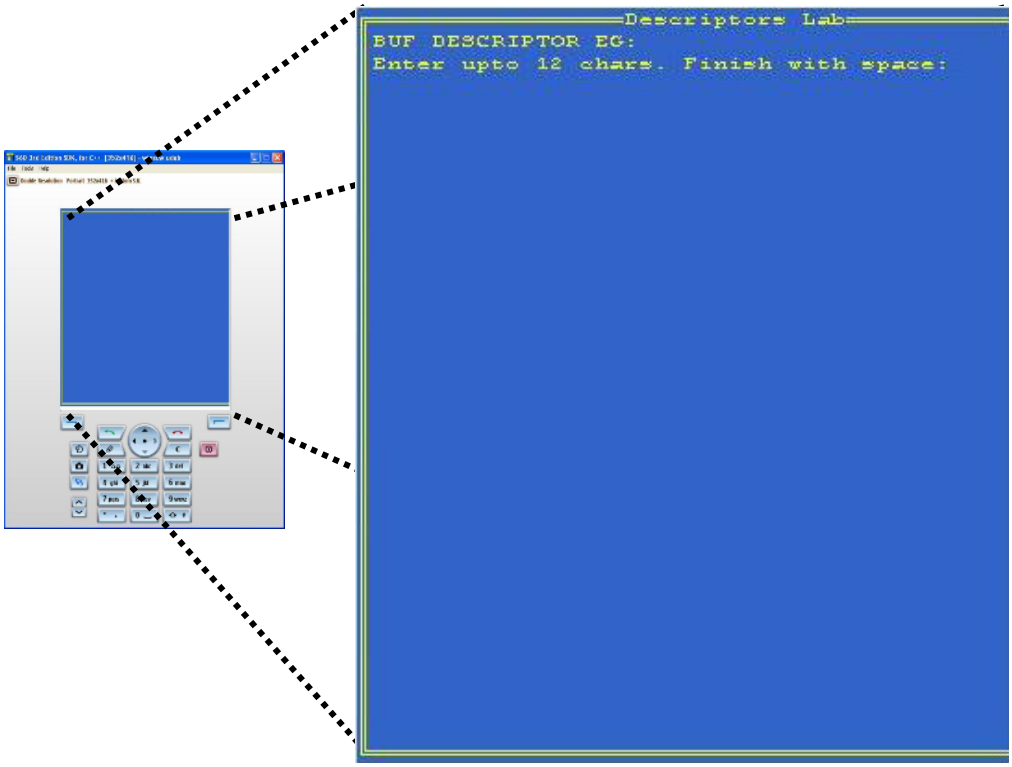


Figure 13 Emulator text window after running the console application

Figure 14 Emulator text window after completing the buffer descriptor example

Figure 15 Emulator text window after completing the heap descriptor example

5. Close the Carbide.c++ IDE. This completes the lab.

Developer
Training

Module #04306

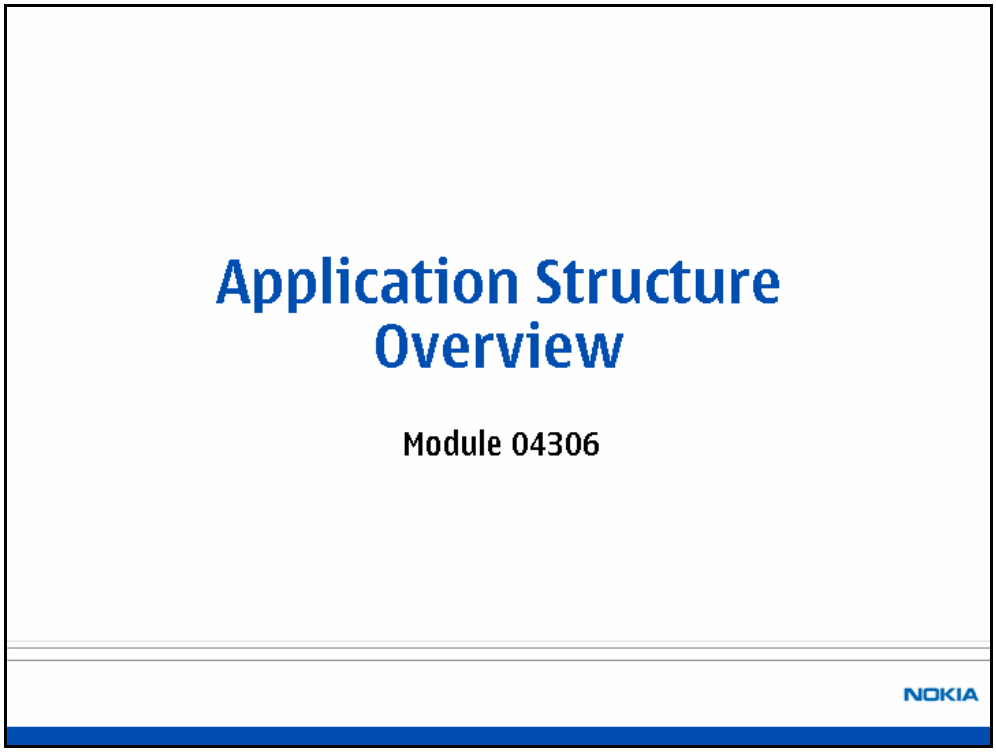
Application Structure Overview

Contents

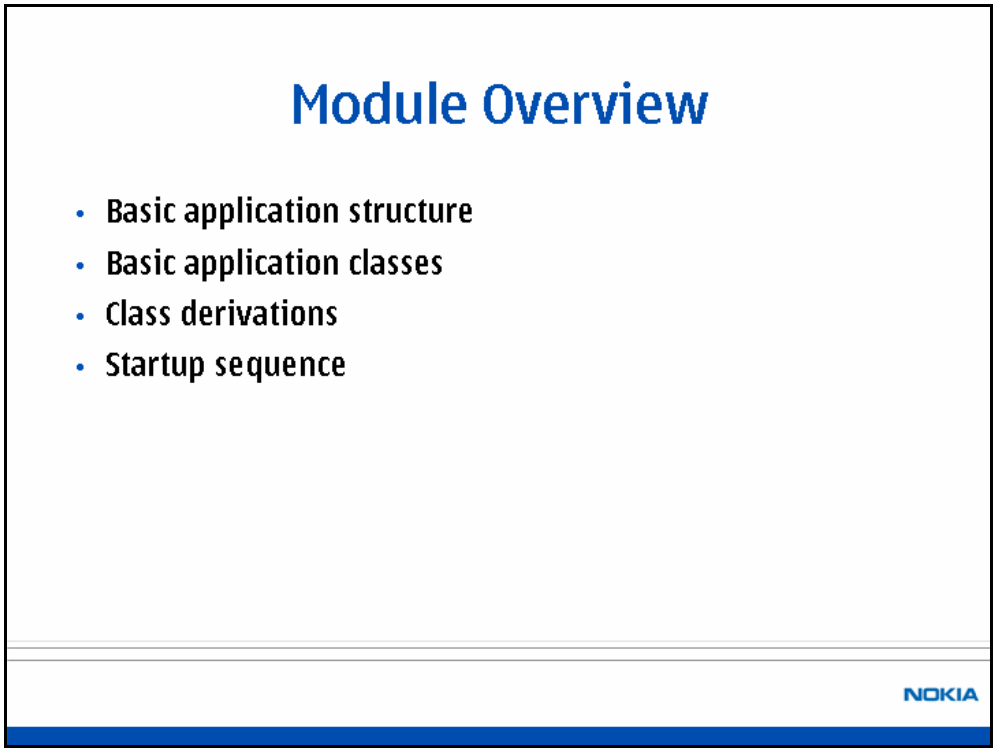
NOKIA

BO-EN-04300-20080108

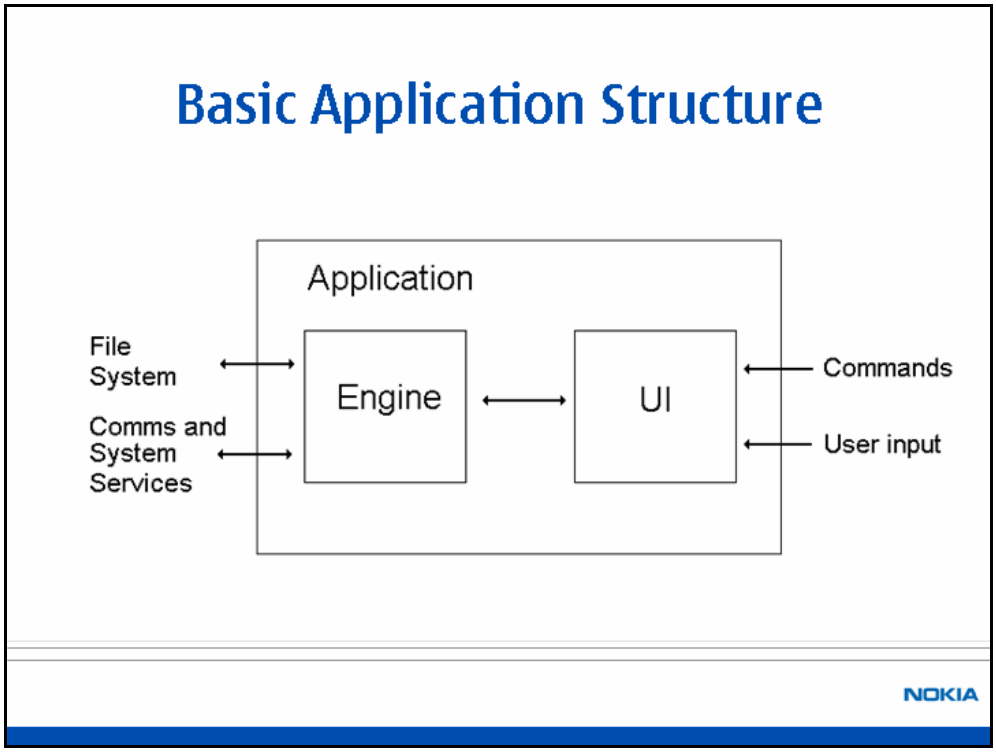
Application Structure Overview



Module Overview

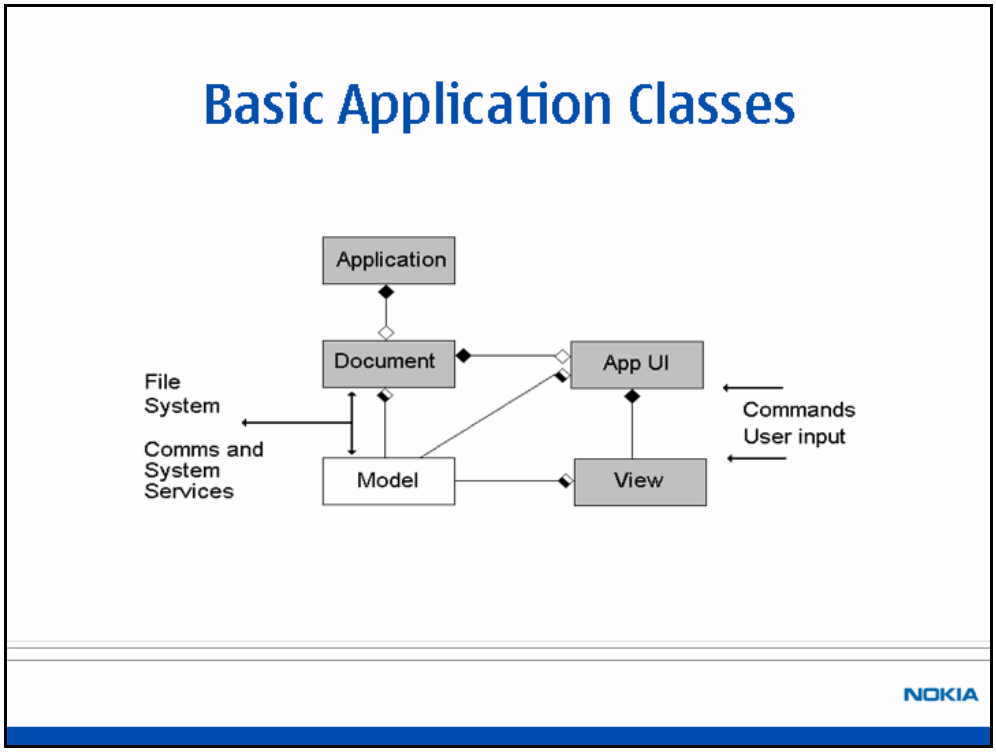


Basic Application Structure



Applications can be split into a UI (also known as a View) and an Engine (also known as a Model). The UI is the part that presents the data to the user. The Engine is concerned with data manipulation and other operations independently of how these are eventually represented to the user. The Engine can therefore be re-used by other applications (it can be built as a shared dll). The example supplied with this training material is too simple to have a standalone Engine, there is not really anything of value to another application. However it is worth bearing in mind, when designing more complicated applications in the future, to separate out the UI independent functionality from your applications for future re-use.

Basic Application Classes

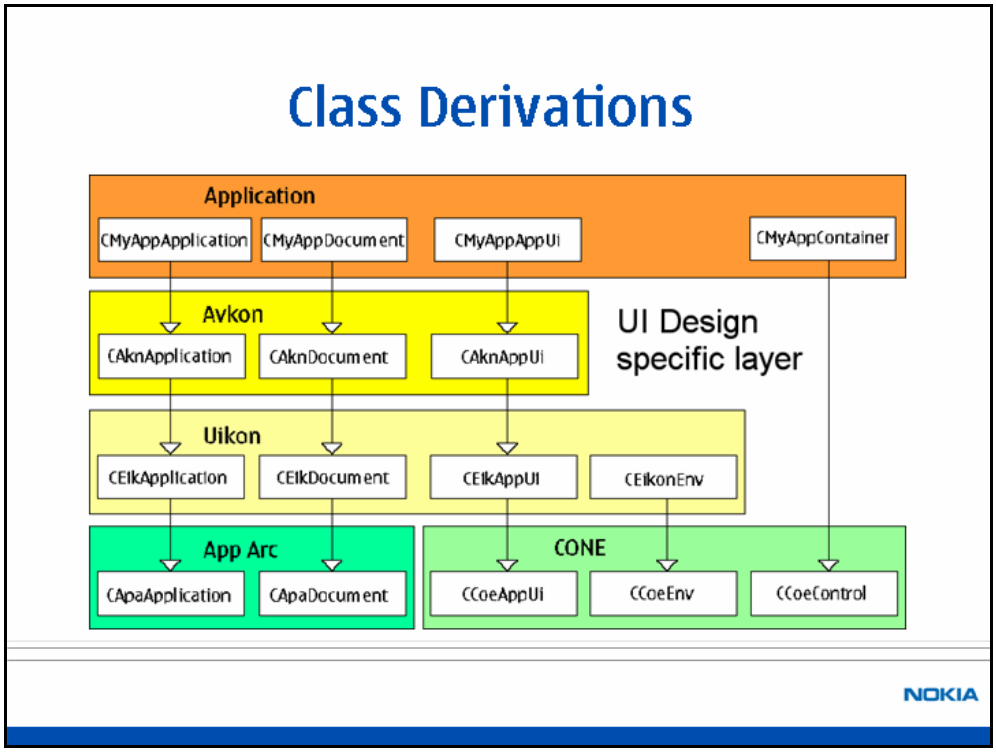


The simple class diagram on the slide shows the classes that make up an application. The Application class provides very little functionality. Its only compulsory responsibilities are to return the UID3 value of the application and to create the Document class.In many S60 applications, the Document class is just used to create the Application User Interface (App UI). However it can also contain functions to read and write state data to a persistent file. However this is disabled by default in S60.

The Application User Interface (App UI) class is the central user interface class. It is not actually visible itself but owns visible controls and handles menu items and key events.The View or Container class contains visible controls that are actually seen by the user. There may be more than one view in an application.

The model is not formally part of the application structure but can potentially be owned by the Document, App UI or View classes depending on the application.

Class Derivations



The slide shows the class derivations used in an example application called MyApp.

All the classes within the top layer of the diagram are user application-specific classes. They are derived from the lower layers to provide the concrete Application instance and the specific functionality.

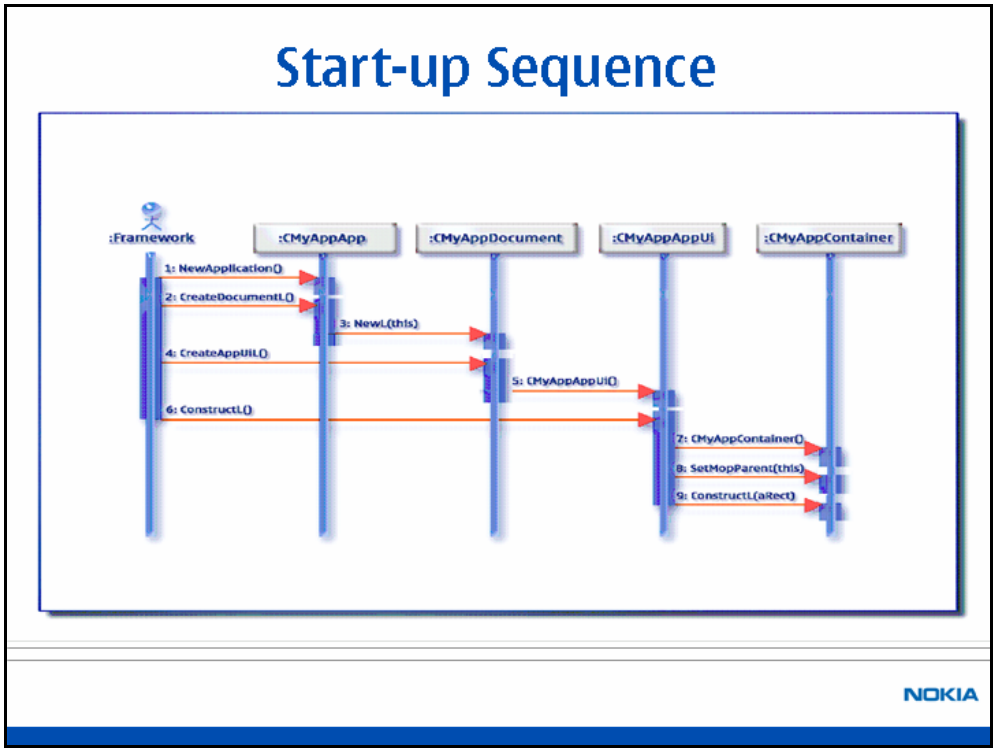
The Avkon classes (prefixed with `CAkn`) in the next layer down are those specific to the S60 application framework. They implement additional S60 specific functionality, over and above the Uikon functionality.

The next layer of classes in the diagram are the Uikon classes (prefixed with `CEik`). These outline the framework of application classes and are present in all UI designs.

The AppArc layer classes (prefixed with `CApa)` define the basic interface for the application classes and additionally for application processes (not shown in the diagram).

The CONE layer classes (prefixed with `CCoe)` specify the base-level visible control class (`CCoeControl`) and additionally the low level asynchronous event handling classes (`CCoeAppUi` and `CCoeEnv`). These are coupled to a dedicated event loop for the application process (not shown in the diagram) that calls virtual functions within `CCoeAppUi` that derived classes override to implement menu item selection and key handling.

Start-up Sequence



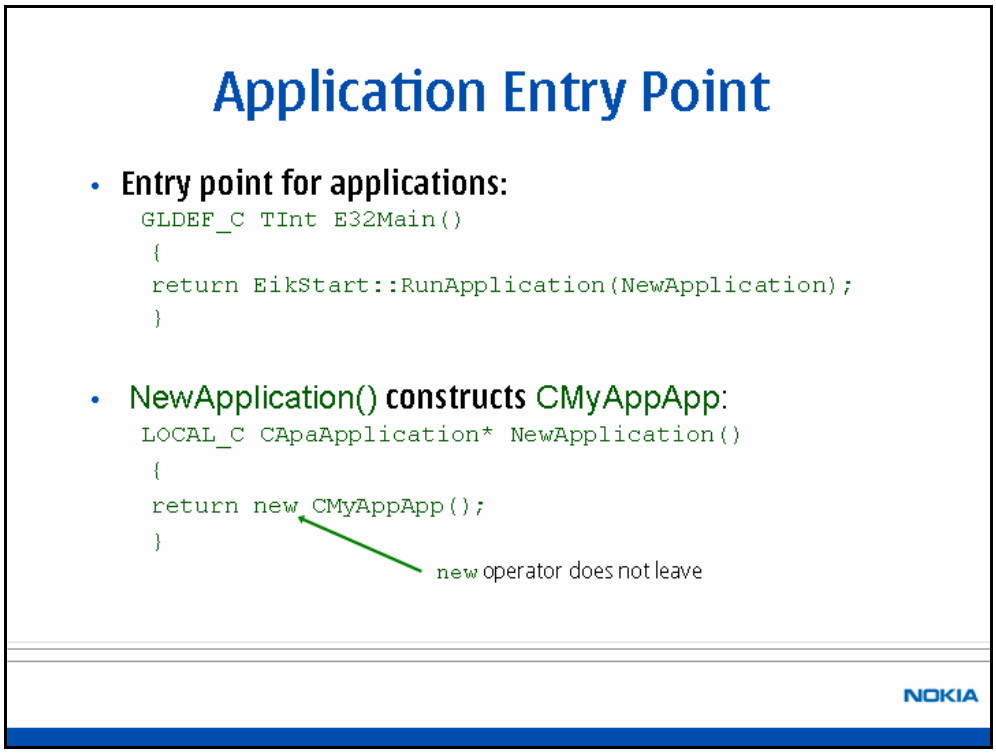
The slide summarises the order of the creation of application framework classes. The underlying framework classes have been simplified into the framework actor. Note that in any case debugging information for these classes is not available with the SDK. If breakpoints are set within the MyApp sample application, the call stack information is not present to say which functions call the application.

When the application process starts, `E32Main()` is called. This is the entry point function for the application. `E32Main()` calls the application framework to run the application. The slide shows the sequence of events from then on. Further explanation of each step in the sequence is included below:

1. The `NewApplication()` function is called by the application framework to create the application object. It is not a member of `CMyAppApp` and cannot leave.
2. `CMyAppApp::CreateDocumentL()` directly calls `CMyAppDocument::NewL()`.
3. First and second phase construction is done here.
4. `CMyAppDocument::CreateAppUiL()` directly calls the default C++ constructor of `CMyAppAppUi`.
5. The default constructor just allocates memory for the object and initialises all members to 0.
6. `CMyAppAppUi::ConstructL()` creates the container.
7. The container object is created here.
8. `CMyAppContainer::SetMopParent()` sets the parent of the container to be the App UI. This is necessary if the container needs to use the scroll bar (scroll bars will be covered in the “UI Features” module).

9. `CMyAppContainer::ConstructL()` is called to perform second-phase construction of the container. Note that a `TRect&` is passed through to the container at this stage. The container is not drawn until `CCoeControl::ActivateL()` is called and control is returned to the active scheduler (the application's event loop). Therefore if any size calculations need to be done at construction time, the area of the container needs to be passed in by the App Ui.

Application Entry Point

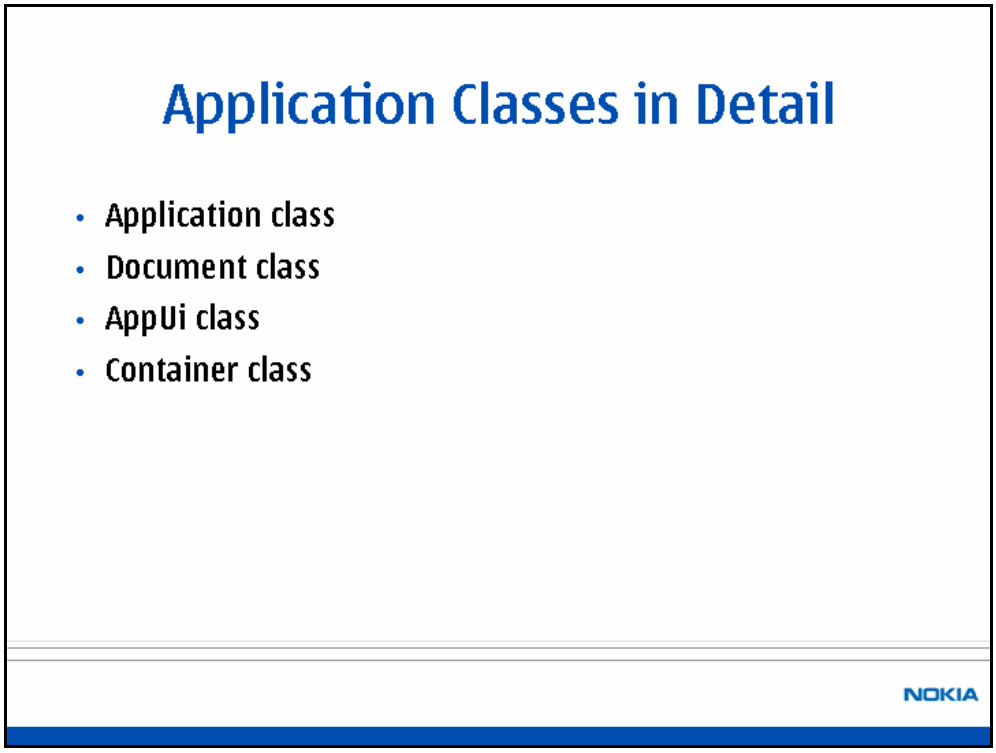


Applications are executables in Symbian OS and run in their own process. Each application must implement the following (global) functions:

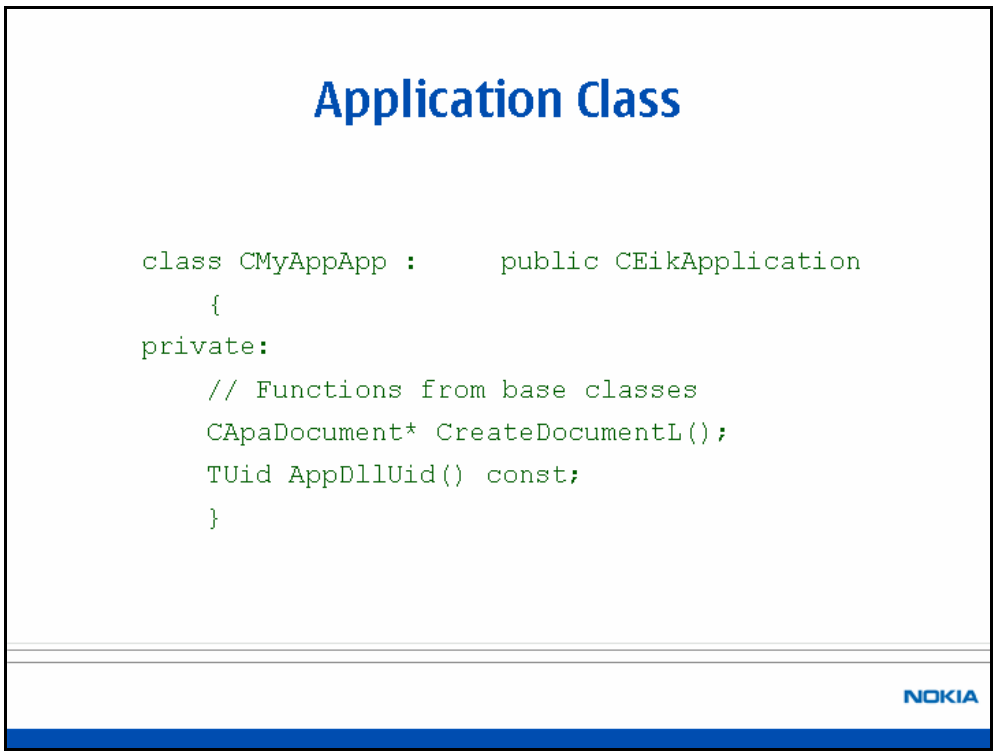
- `E32Main()`: This function is the main entry point for an application (or indeed any executable). For applications, its primary use is to call the framework function `EikStart::RunApplication()` to initialise and run the application.
- `NewApplication()`: This function is also required so that the application process can create an instance of the application class. The returned application instance must be derived from `CApaApplication`, an abstract class that defines the features all application classes must support.

Note that the above information only applies to applications written for Symbian v.9.x and later. In previous versions of Symbian OS, applications were implemented as polymorphic DLLs and ran within a single application framework process.

Application Classes in Detail



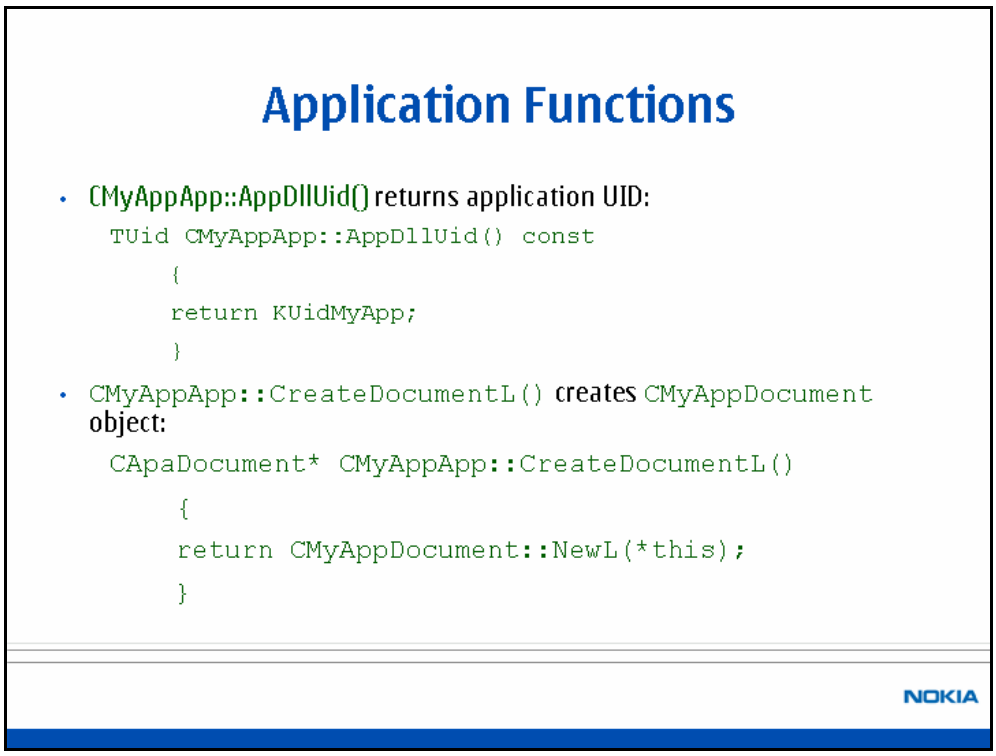
Application Class



The application class is the first class that is constructed when an application launches. All application classes are derived from `CApaApplication`. This class defines the basic behavior for applications. An application class has two roles: as a factory that creates concrete document objects, and as a supplier of utility functions not specific to any particular instance of a document. The utility functions include returning an application's caption, or accessing its .ini file.

The Uikon framework derives from `CApaApplication` to give `CEikApplication`, which provides an interface to the resource file and the document.

Application Functions

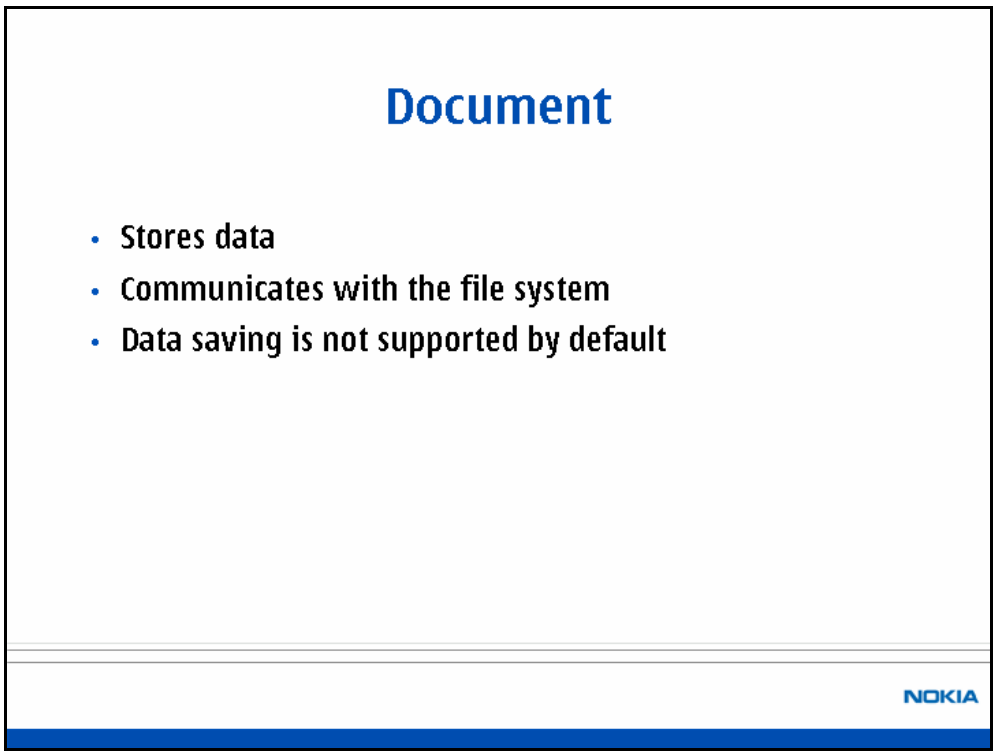


`AppDllUid()` and `CreateDocumentL()` are defined as pure virtual functions in `CApaApplication`, which is the base class for all applications. `CEikApplication` does not implement `AppDllUid()` and `CEikApplication::CreateDocumentL()` creates a default Uikon document object so these functions must be provided by the user application class.

`AppDllUid()` is called by the application process immediately after the application creation and returns the application specific UID3. This UID is used to differentiate between different applications.

`CreateDocumentL()` is called by the application process when a new document is required. The document is added to the application process' list of documents (for all applications). It is always called after `AppDllUid()` upon application creation.

Document

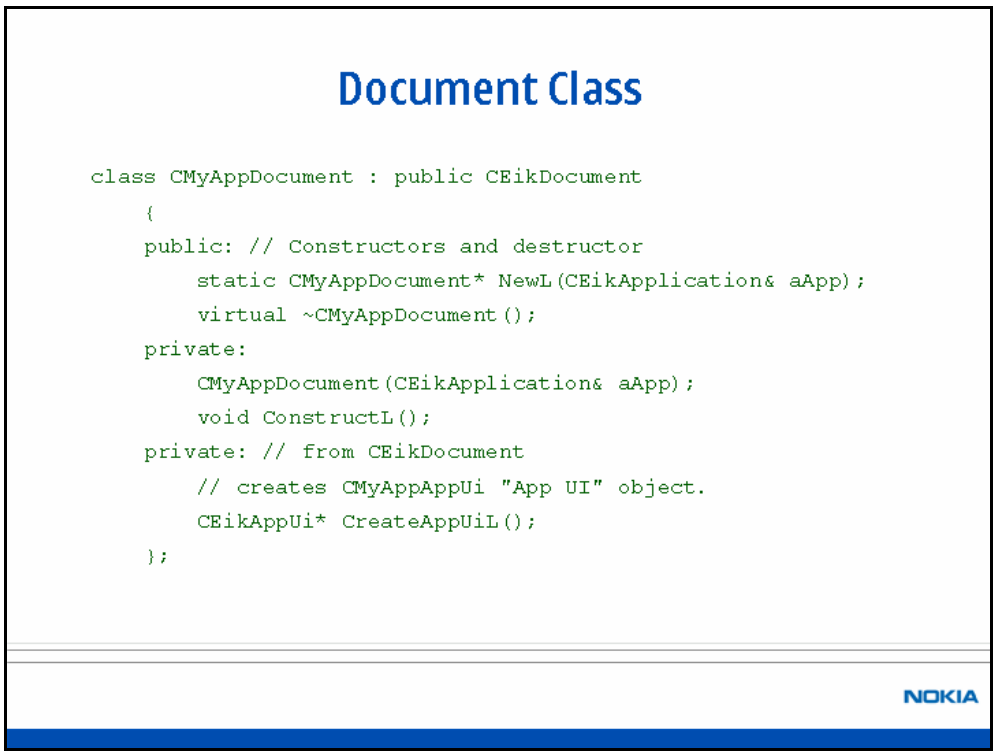


The document class is traditionally used to store application data to file. This enables applications to be persistent. For example, a calendar application will need to be persistent, so that reminders can be entered by the user and read at another time. Additionally document classes are used to construct the App UI (see below).

Symbian OS provides an abstract base document class, `CApaDocument`, which specifies several pure virtual functions that derived classes must implement. This defines the basic behavior for document classes.

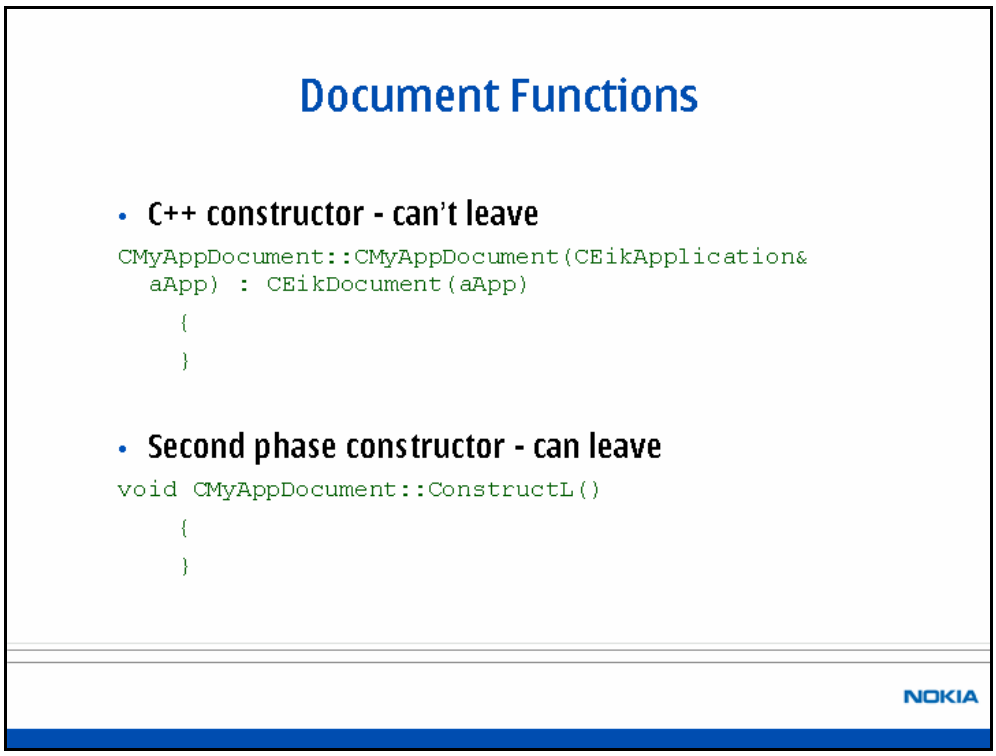
The Uikon framework provides `CEikDocument`, which derives from `CApaDocument` and implements the core document file support.

Document Class



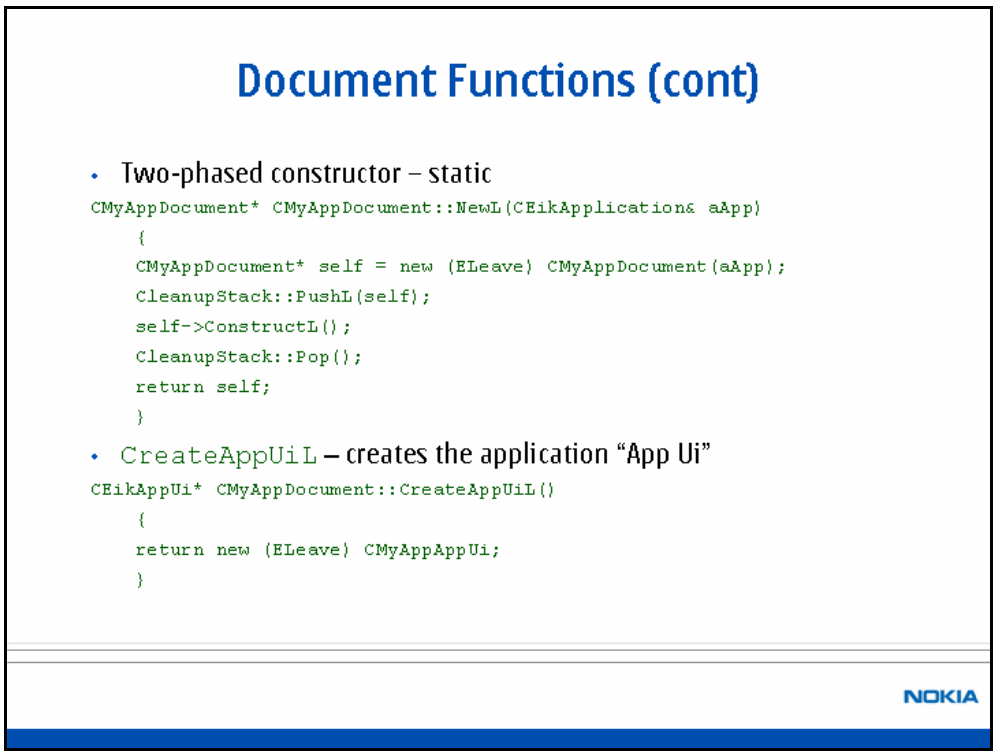
The example on the slide does not require document file support, so the user document class `CMyAppDocument` derives from `CEikDocument`. Note that since this application does not save anything to file, there are no data members. The only functions implemented are the constructors, destructor and `CreateAppUiL()`, which creates the application user interface class instance (App Ui). The following two slides cover these functions in more detail.

Document Functions



The user document class implements both a default C++ and a second phase constructor. As you can see from the slide, the second phase constructor is empty since there is no data to allocate dynamically. However, it is included because in the future there may be dynamically allocated data owned by the document class.

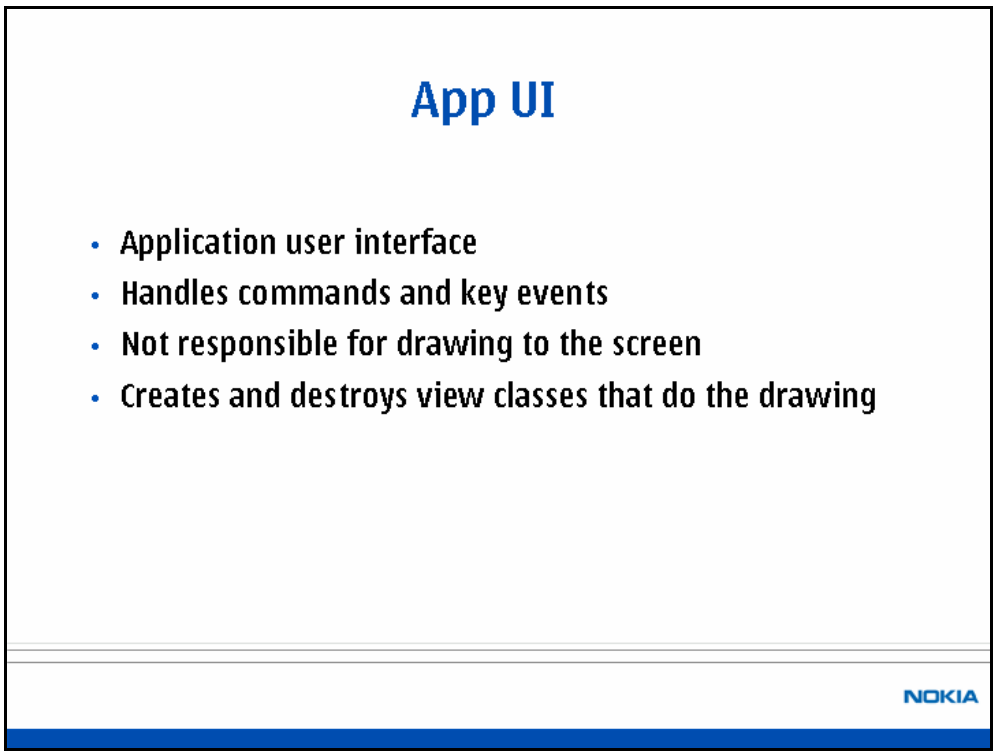
Document Functions (cont)



A static two-phase constructor is provided. This is called by `CMyAppApp::CreateDocumentL()`.

The `CreateAppUiL()` function creates the application user interface ("App UI"). For non-file based applications such as the example shown, the document class will not be used for anything else.

App UI



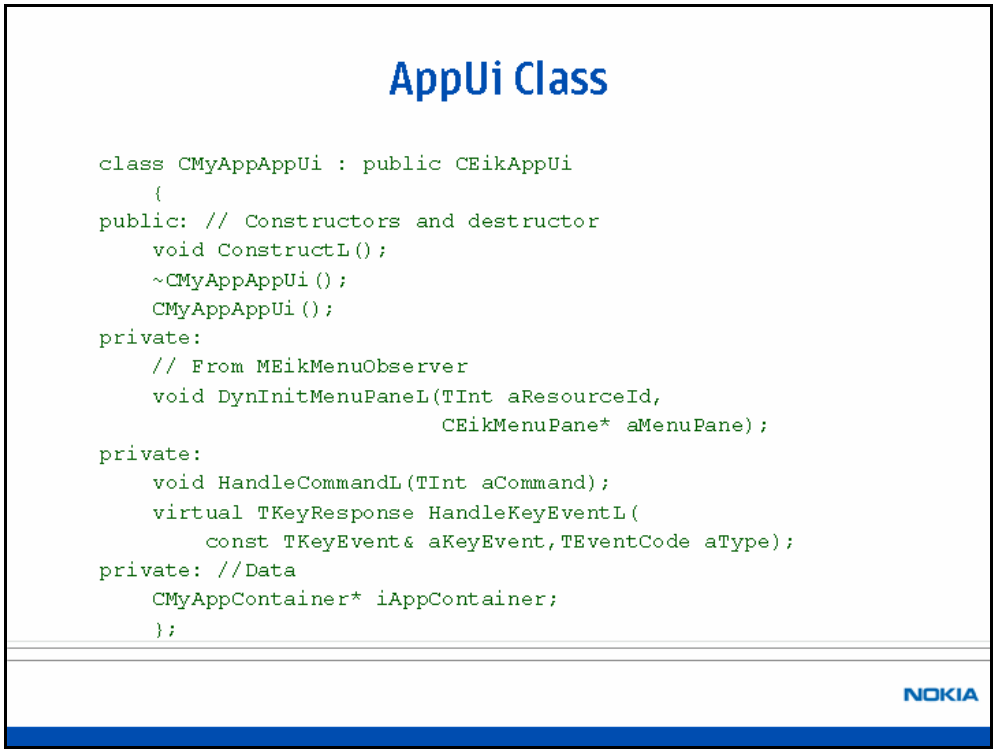
The application user interface is the central user interface class. It creates and owns controls to display the application data, and centralizes handling of command input from standard controls such as menus and toolbars. The App UI is not visible and delegates drawing to the controls it contains.

For file-based applications, the application UI additionally accesses the document class in order to store the application data in response to user commands.

Responses to various kinds of events can be handled by providing suitable implementations of the following virtual functions:

- `HandleKeyEventL()`: Key events.
- `HandleForegroundEventL()`: Application switched to foreground.
- `HandleSwitchOnEventL()`: Device switched on.
- `HandleSystemEventL()`: System events.
- `HandleWsEvent()`: Window server events.
- `HandleApplicationSpecificEventL()`: Application-specific events.
- `HandleCommandL()`: Handles commands defined in resource files.

AppUi Class



It can be seen from the slide that there is a default C++ constructor and a second-phase constructor as expected. However, there is no two-phase constructor. This is not needed since the first phase of construction is done from `CMyAppDocument::CreateAppUiL()`. The second phase constructor is virtual and this overrides `CEikAppUi::ConstructL()`. It is called from the application framework.

`CMyAppAppUi::DynInitMenuPaneL()` is called by the framework just before it displays a menu pane. The application can set the state of menu items dynamically, according to the state of the application data.

`CMyAppAppUi::HandleKeyEventL()` is called if a key event occurs that has not already been handled by any of the controls on the control stack. If the key event is handled by any of the controls owned by the application UI, the return value should be `EKeyWasConsumed`. Otherwise, `EKeyWasNotConsumed` should be returned.

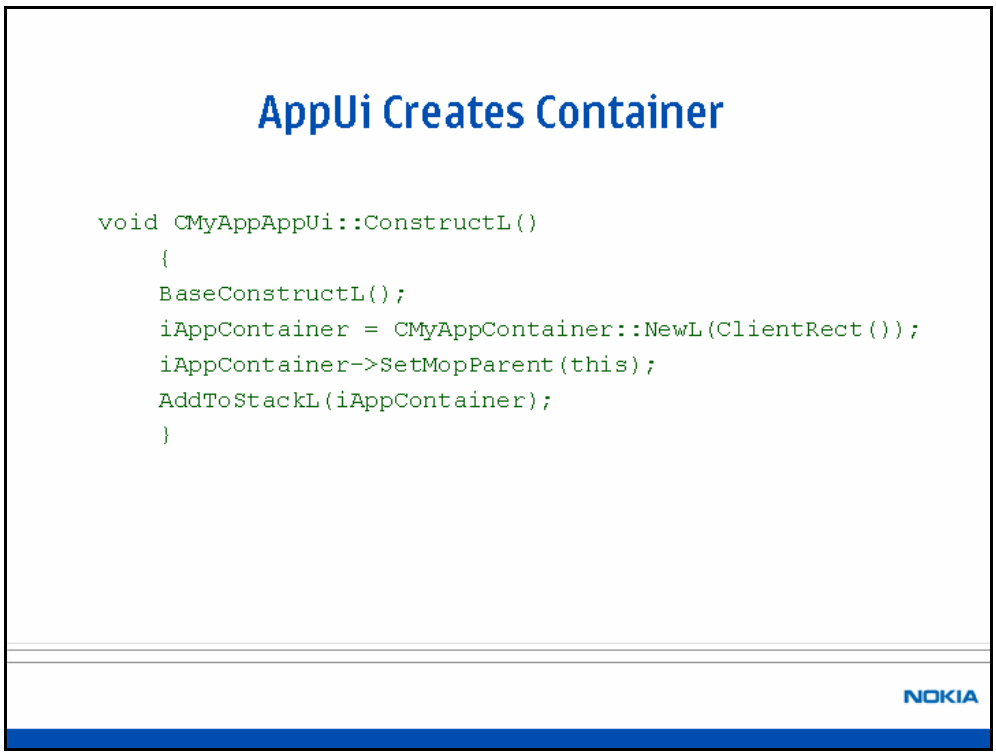
Handling Commands

The handling of user-commands, such as menu item selections, is done in `CMyAppAppUi::HandleCommandL()`.

The command ID is passed to the function and appropriate action can be taken depending on the value. The function is called from the application framework because it overrides `CEikAppUi::HandleCommandL()`. In the example shown on the slide, the command ID is `EEikCmdExit` when the framework informs the application to shut down. `CEikAppUi::Exit()` is called to shut down the application. These two command IDs are system defined.

`EMyAppCmdAppTest` is a user-defined command launched from the popup-menu. The enum is defined in `MyApp.hrh` and the menu is defined in the `MyApp.rss` application resource file. Resource files will be covered later in this course. When the menu item associated with `EMyAppCmdAppTest` is selected, a test message is displayed on the screen. This remains on the screen for a few seconds before disappearing.

AppUi Creates Container

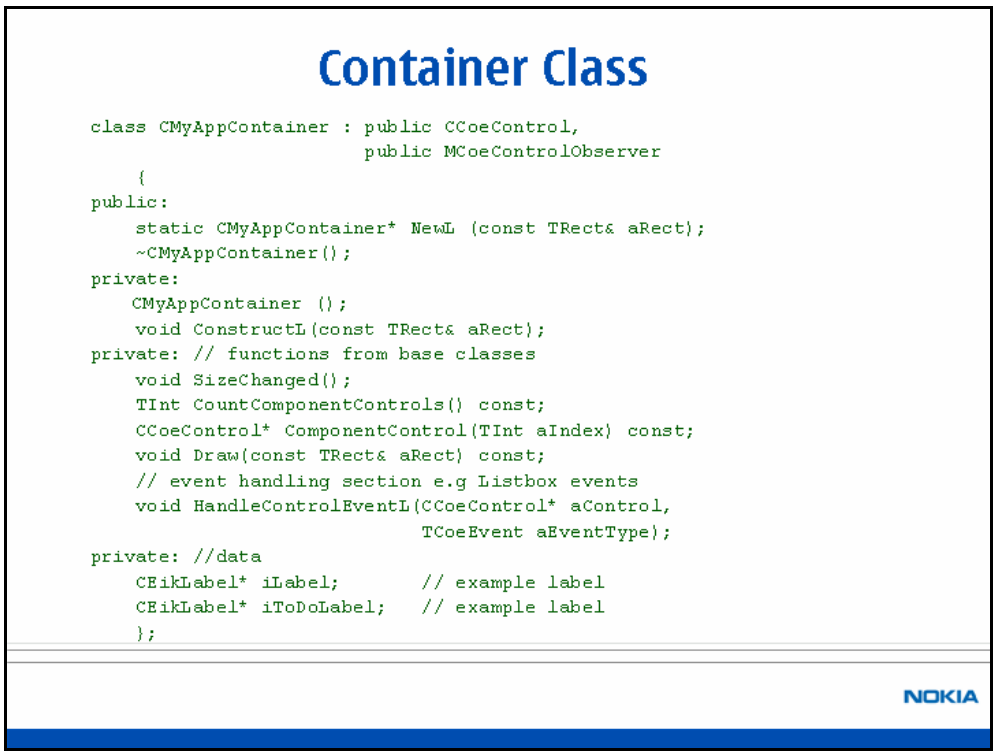


A container is created in the second-phase constructor of the App UI on which to place controls. By S60 convention, all controls are called containers even if they do not contain multiple controls.

The call to `CCoeControl::SetMopParent()` enables the container to access the App UI so that any scroll bars can be displayed and updated. The call to `CCoeAppUi::AddToStack()` adds the container onto the App UIs control stack. The control stack is a mechanism for handling key press events and contains a list of controls that wish to receive keyboard events. In the case of the example shown on the slide, the container does not need to receive key events. However, adding it onto the control stack merely enables the functionality for future use. If a control wished to handle key events, it would need to override `CCoeControl::HandleKeyEventL()` and `CCoeControl::InputCapabilities()`.

The call to `CMyAppContainer::NewL (TRect& aRect)` performs complete construction of the container and passes the area for drawing through to it.

Container Class



The container is a composite control and can contain any number of controls. In the example shown, there are two labels. These display static text to the user (although they can be changed programmatically).

Note the standard use of `NewL()` to construct the container, and the fact that the `ConstructL()` and C++ constructor are private.

The following `CMyAppContainer` functions are overrides of `CCoeControl` functions:

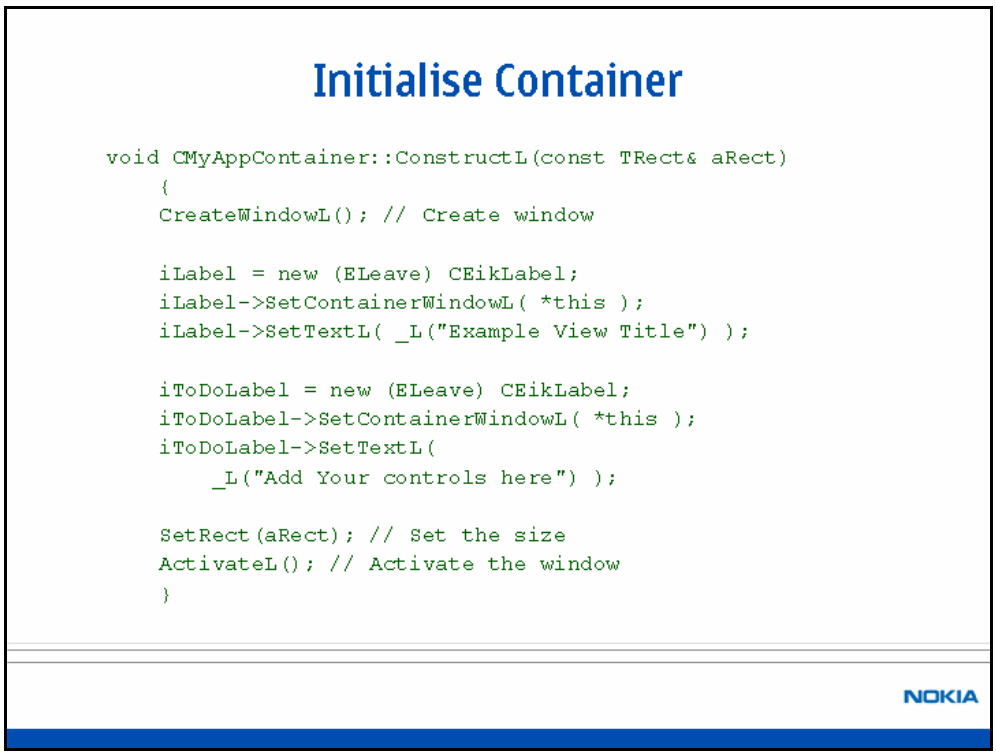
- `SizeChanged()`: Called when the container is created/resized - sizes the labels to their minimum possible size.
- `CountComponentControls()`: Returns the number of controls owned by the container – 2 in this case.
- `ComponentControl()`: Returns a pointer to a control owned by the container when passed a zero based index. NULL is returned if the control for the index does not exist.
- `Draw()`: Performs some container specific drawing. In this case draws a black bordering rectangle around the edge of the container area of the screen.

The following `CMyAppContainer` function is an override of `MCoeControlObserver`:

- `HandleControlEventL()`: This is called when a control owned by the container calls `CCoeControl::ReportEventL()`. This does not occur in any `CMyAppContainer` controls so the function implementation is empty. It is quite rare that this function needs to be overridden. An example of when it could be overridden is when the state data of a control changes and it needs to notify its parent container.

The member data `iLabel` and `iToDoLabel` are pointers to label controls that display text on the screen.

Initialise Container



All the container initialisation is done in the second phase constructor.

Firstly, the window for the container is created with a call to `CCoeControl::CreateWindowL()` – this is necessary so that the container and its controls are displayed to the user. Note that it is only necessary to create a window for each top-level container and not one for each control.

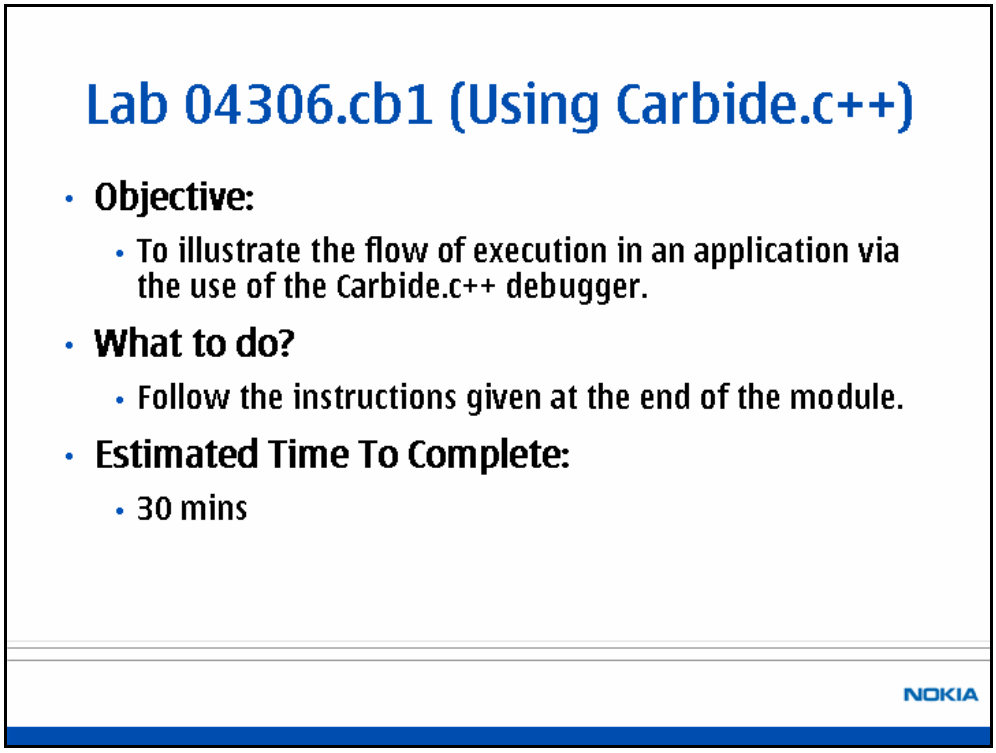
The labels are set up in three steps:

- They are constructed using the default C++ constructor.
- They are attached to the container window. This is done by calling `CCoeControl::SetContainerWindowL()` and is needed since the labels do not own a window themselves, so they need a reference to the container object which has created a window.
- The text of each label is set. In production quality code, this should originate from the localisation file that is associated from the resources. This will be covered in the resources section of the module.

Next, the extent of the container is set, by calling `CCoeControl::SetRect()`, using the area passed as a parameter to the constructor. This will clip the labels if they extend outside of the area.

Finally, the container signals that it is ready to be drawn by calling `CCoeControl::ActivateL()`. This is called at the end of the function so that the container's contents are drawn after everything else has been set up.

Lab 04306.cb1 (Using Carbide.c++)



Lab 04306.cb1

Overview

Title:	Debug an application to understand its flow of execution		
Overview:	This lab demonstrates the order of creation of the standard application objects: Application object, Document object, AppUi object and Container object. The lab begins by creating a new application using the built in Carbide.c++ wizard. For this reason no starter code is provided. The lab then uses debugging to show the order of creation and destruction of the application objects. As no code changes are performed in this lab no solution code is provided.		
Objectives:	To understand how to: • Generate a new S60 application using the built in Carbide.c++ wizard. • Use the Carbide.c++ debugger. • Explain the order of construction and destruction of standard application objects.		
Compatible IDE(s):	Carbide.c++ v1.2		
Compatible SDK(s):	S60 3rd Edition, S60 3rd Edition (MR), S60 3rd Edition FP1		
App. Type:	Standard GUI	**App. Name:**	MyApp.exe
Starter Code Provided:	No	**Solution Code Provided:**	No
Estimated Time To Complete:	30 minutes		

Lab Instructions

Exercise 1 – Preliminary Steps

Task 1.1 – Checks

1. Ensure that a compatible SDK has been installed. A list of compatible SDKs to use with this lab is given in the table in the Introduction.
2. Ensure that a compatible IDE has been installed. A list of compatible IDEs to use with this lab is given in the table in the Introduction.

Task 1.2 – Make sure the emulator is not in text shell mode

1. Open Windows Explorer.
2. Navigate to the file named epoc.ini, located in the `<EPOCROOT>\epoc32\data` folder. Here, `<EPOCROOT>` is a placeholder for the root folder location of your SDK.
3. Open the epoc.ini file using Notepad.

4. Check that the file does not contain the “textshell” statement. If the statement is present in the file, then either delete the entire line containing the statement or comment the statement out. The latter is done by inserting a # symbol at the start of the line containing the statement.
5. Press ‘Ctrl’ + ‘S’ to save the contents of the file (if you altered it) and close Notepad.
6. Close Windows Explorer.

Task 1.3 – Carbide Setup

1. Launch the Carbide.c++ IDE. Depending on your Carbide.c++ settings the IDE will either display the “Workspace Launcher” dialog or open the last used workspace. If the latter occurs open the “Workspace Launcher” dialog by selecting the “File -> Switch Workspace...” menu item.
2. In the workspace launcher window, create a new workspace by typing `C:\Labs\<Workspace>` in the Workspace field. Here `<Workspace>` is a placeholder for a folder name of you choosing that does not already exist and does not contain spaces.
3. Click the OK button.
4. Close the Welcome window after it appears when the workspace is opened.

Task 1.4 – Creating a new project

Note there are no starter files (or solution files) associated with this lab. Instead a project is generated using one of the application generation wizards built in to Carbide.c++.

1. Select the “File -> New -> Project” menu item. This launches the new project dialog.
2. Select “Symbian OS c++ Project”
3. Select “3rd Ed Gui Application” from the tree list.
4. On the first screen of the dialog enter “MyApp” in the “Project name” field and click the Next button.
5. You are now asked to select the S60 SDK build configurations your project should work with. All configurations for your chosen compatible SDK should be selected. When this is done a tick appears in the check box alongside each configuration.
6. Accept the selected configurations and the rest of the default settings by clicking the Finish button. The wizard generates all the necessary project files. The MyApp project appears in the “C/C++ Projects” view.

Exercise 2 – Understanding an application’s flow of execution

Task 2.1 – Setting breakpoints in the application

This task will set breakpoints in the application in order to illustrate the sequence of events that occur when an application is launched and run.

1. Expand the `/MyApp/src` folder node in the “C/C++ Projects” view. A list of the MyApp project’s source code file-nodes are displayed.
2. Double click on the `MyApp.cpp` source file node in “C/C++ Projects” view. A text editor window is opened showing the contents of the file.
3. Expand the `MyApp.cpp` source file node in “C/C++ Projects” view. Among other items, the functions contained in source file are displayed as nodes.
4. Double click the `E32Main` function node in “C/C++ Projects” view. The cursor in the text

editor window moves to the function name in the source file.

5. On the line containing the opening brace of the `E32Main` function, set a regular breakpoint by double clicking the marker bar to the left of the line of code.
6. Note that as an alternative method for setting breakpoints, you can right-click on the marker bar to open its context menu and select the "Toggle Breakpoint" menu item. A circle is displayed in the marker bar, alongside a particular line of source code, when a breakpoint has been set for that line.
7. Double click the `NewApplication` function node in "C/C++ Projects" view. The cursor in the text editor window moves to the function name in the source file.
8. On the line containing the opening brace of the `E32Main` function, set a regular breakpoint as described above.
9. Close the text editor window containing the `MyApp.cpp` source code.
10. Following the previous steps for setting breakpoints in `MyApp.cpp`, set a breakpoint on the line containing the opening brace for each of the following functions in `MyAppApplication.cpp`:
    - `CMyAppApplication::CreateDocumentL`
11. Also, set a breakpoint on the line of code containing the opening brace for each of the following functions in `MyAppAppUi.cpp`:
    - `CMyAppAppUi::ConstructL`
    - `CMyAppAppUi::~CMyAppAppUi`
    - `CMyAppAppUi::HandleCommandL`
12. Set a breakpoint on the line of code containing the opening brace for each of the following functions in `MyAppAppView.cpp`:
    - `CMyAppAppView::NewL`
13. Finally, set a breakpoint on the line of code containing the opening brace for each of the following functions in `MyAppDocument.cpp`:
    - `CMyAppDocument::NewL`
    - `CMyAppDocument::CreateAppUiL`
    - `CMyAppDocument::~CMyAppDocument`

Task 2.2 – Building the application

1. Build the application for the Emulator Debug configuration for the SDK you are using by right clicking on the MyApp project node in "C/C++ Projects" view and select the "Build Project" or "Rebuild Project" menu item.

Task 2.3 – Debugging the application

1. Right click on the MyApp project node in "C/C++ Projects" view and select the "Debug As > Debug Symbian OS Application" menu item. The emulator will launch in debug mode and the Carbide.c++ IDE will switch to its Debug perspective.
2. Follow the instructions for this step that are specific to the SDK you are using:

   S60 3rd Edition SDK and S60 3rd Edition MR SDK

   After the emulator has booted the Applications menu is shown. The MyApp application is located in the "Installat." folder. Navigate to this folder using the navigation keys and open it using the selection key.

S60 3rd Edition FP1 SDK

After the emulator has booted the Standby application is shown. Switch to the Applications menu by selecting the Applications key. The MyApp application is located in the "Installed" folder. Navigate to this folder using the navigation keys and open it using the selection key.

3. Navigate to the MyApp application, again using the navigation keys, and launch it using the selection key. Execution halts at the first encountered breakpoint and the Windows OS application focus switches back to the Carbide.c++ IDE.

   In the Debug perspective, the Debug view shows the target information in a tree hierarchy. As indicated in this window the MyApp application thread has been suspended because a breakpoint has been hit.

   The child nodes of the suspended thread represent its stack frames. The top stack frame is for the `E32Main` function which is located in `MyApp.cpp`. This is the function in which the first breakpoint is hit. This is unsurprising as `E32Main` is the entry point function for S60 3rd Edition applications.

   Also in the Debug perspective, a text editor window displays the source code for the `E32Main` function. Notice that the line that we set a breakpoint on is highlighted in green indicating that execution has halted there.

   See the Carbide.c++ Help for more information about the Debug perspective and Debug view.

4. Click the Resume toolbar button in the Debug perspective. To find out which button is the Resume button, hover over a button and wait for the tooltip to appear. Execution continues until the next breakpoint is hit.

   The next breakpoint to be hit is the one in the `NewApplication` function which is located in `MyApp.cpp`. This function creates the main application class, `CMyAppApplication`, and returns it to the application framework.

5. Click the Resume toolbar button to begin execution again. Execution is halted when the breakpoint in the `CMyAppApplication::CreateDocumentL` function is hit. This function is located in `MyAppApplication.cpp`. This function is called to create the application's document class.

6. Click the Resume toolbar button. The next breakpoint that is hit is in the `CMyAppDocument::NewL` function which is located in `MyApp.cpp`. This function is called to perform two-phase construction of the document class.

7. Click the Resume toolbar button. The next breakpoint that is hit is in the `CMyAppDocument::CreateAppUiL` function which is located in `MyAppDocument.cpp`. This function is called by the application framework and performs the first phase of construction of the application's UI class.

8. Click the Resume toolbar button. The next breakpoint that is hit is in the `CMyAppAppUi::ConstructL` function which is located in `MyAppAppUi.cpp`. This function is called by the application framework and performs the second phase of construction of the application's UI class.

9. Click the Resume toolbar button. The next breakpoint that is hit is in the `CMyAppAppView::NewL` function which is located in `MyAppAppView.cpp`. This function constructs the application's view.

10. Click the Resume toolbar button. The application completes its initialisation and the MyApp application is now visible on the emulator screen.

11. On the emulator select the Options softkey and highlight the Exit menu item.

12. Press the Return key to select the Exit menu item and exit the application. The code halts at the breakpoint in the `CMyAppAppUi::HandleCommandL`, which is located in `MyAppAppUi.cpp`. This function handles commands passed on by the application

framework; in this case the function has been called to handle the `EAknSoftkeyExit` command. The function handles this command by calling the Exit function.

13. Click the Resume toolbar button. The next breakpoint to be hit is in the `CMyAppAppUi` destructor which is located in `MyAppAppUi.cpp`. This function deletes the application's view.
14. Click the Resume toolbar button. The next breakpoint to be hit is in the `CMyAppDocument` destructor which is located in `MyAppDocument.cpp`.
15. Click the Resume toolbar button again. The application exits after which the Applications menu is displayed.
16. Close the emulator.
17. Close the Carbide.c++ IDE. This completes the lab.

Developer
Training

Module #04307

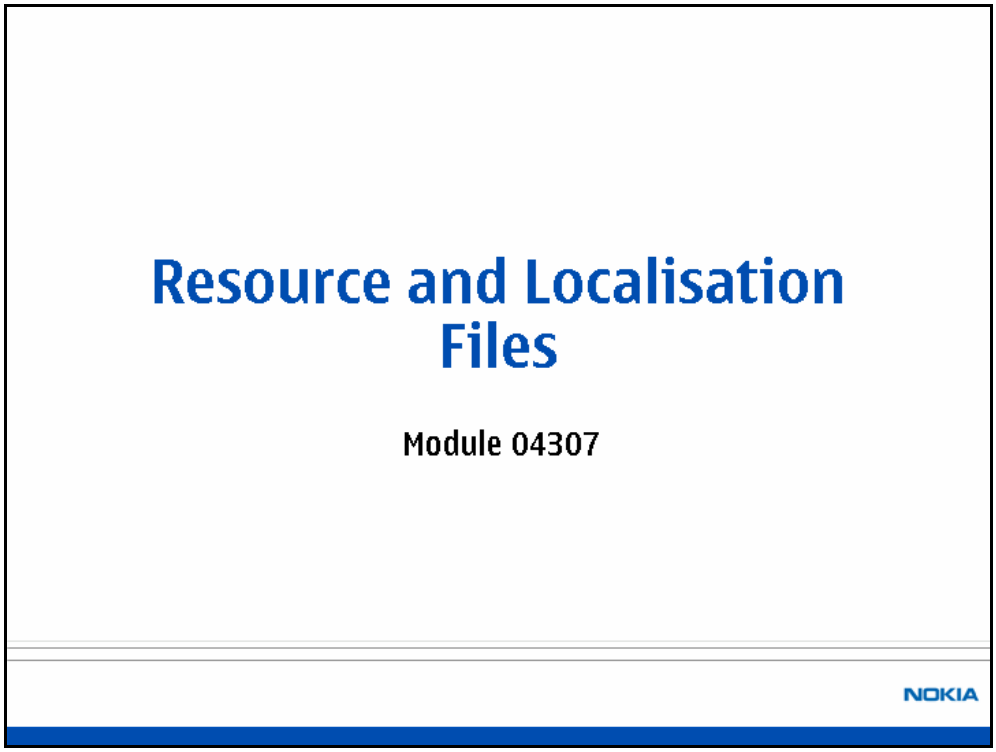
Resource and Localisation Files

Contents

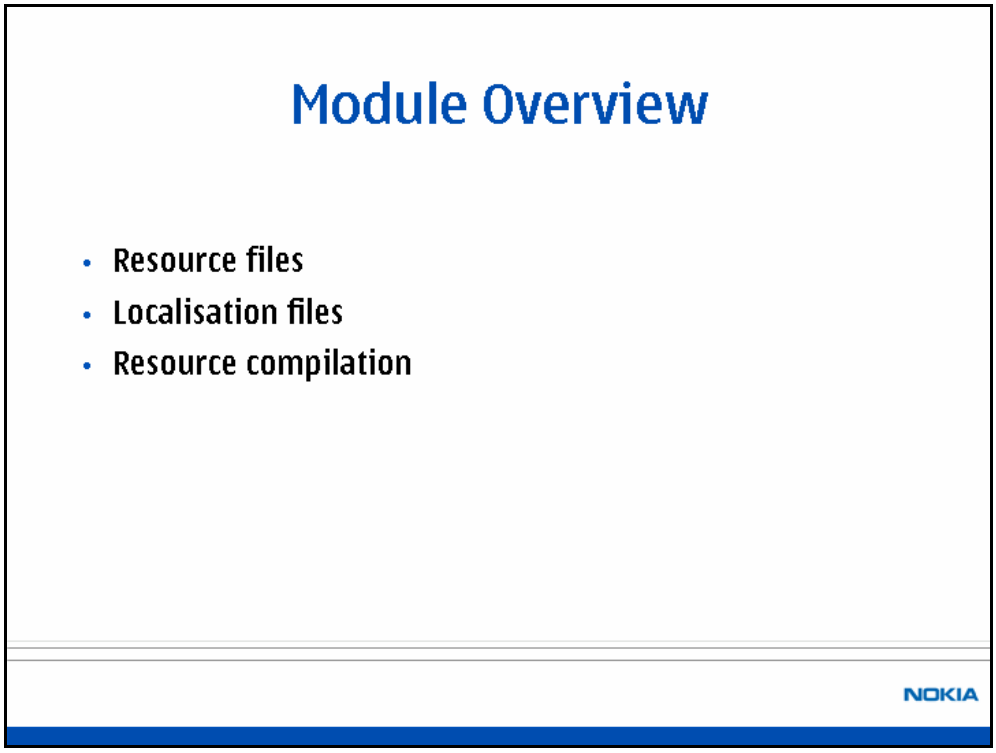
NOKIA

BO-EN-04300-20080108

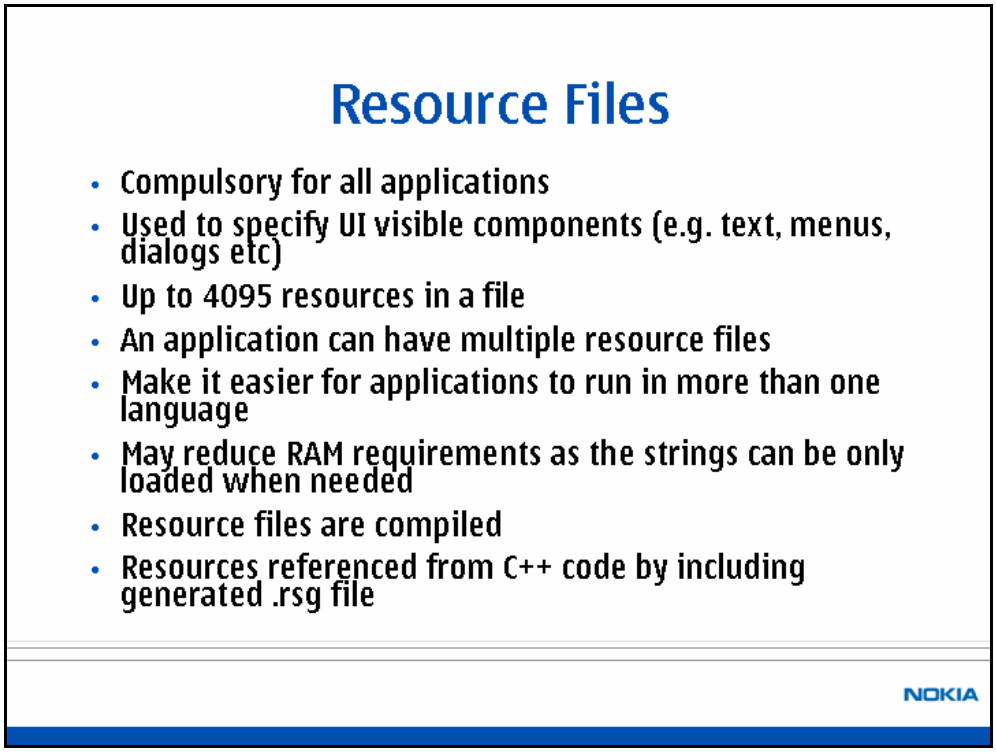
Resource and Localisation Files



Module Overview



Resource Files



A resource file is compulsory for all applications. It specifies information about the application and can contain UI components that are to be displayed.

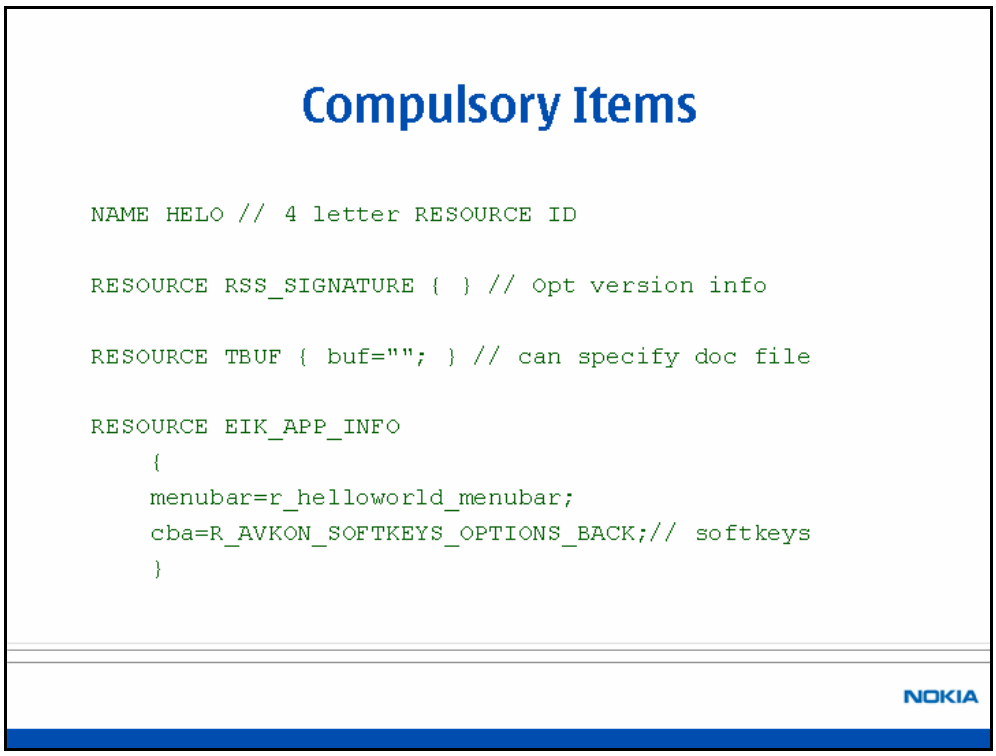
Resource files have the extension of .rss and the default resource file for an application is `<Application Name>.rss`. For example, for a project named “HelloWorld its default resource file would be called `HelloWorld.rss`.

It is possible to load additional resource files dynamically at run-time and access the resources contained within them.

Uikon predefines various resource structures that can be used within the resource file. Different UI designs usually add to these definitions to tailor UI components for the look and feel of the UI.

The most effective way of covering resource files is to go through `HelloWorld.rss` and explain each code fragment one by one. This is done in the following three slides.

Compulsory Items




```
NAME    MYAP // 4 letter ID

#include <eikon.rh>
#include <avkon.rsg>
#include <avkon.rh>
#include <avkon.mbg>
```

The `NAME` item allows an application to access multiple resource files. It has to differ from the names used by the Uikon and S60 system resource files. The four `#include` statements load resource structure definitions that are used in the resource file.

```
#include "HelloWorld.hrh"
#include "HelloWorld.loc"
```

The header file, `HelloWorld.hrh`, defines enumerated constants for the application's commands. There is only one value defined since there is only one application specific menu item. This is `EHelloWorldCmdAppTest`. See the "Menu Pane" slide to see how this value is used.

The header file, `HelloWorld.rls` (.loc file in versions of Symbian OS before v.9.x), defines strings that are used in the UI. They need to be translated into different target languages for localisation purposes. This topic is discussed in the next section.

After the `NAME` and the `#include` lines, every resource file should include compulsory information about the application:

```
RESOURCE RSS_SIGNATURE { }

RESOURCE TBUF { buf=""; }

RESOURCE EIK_APP_INFO
```

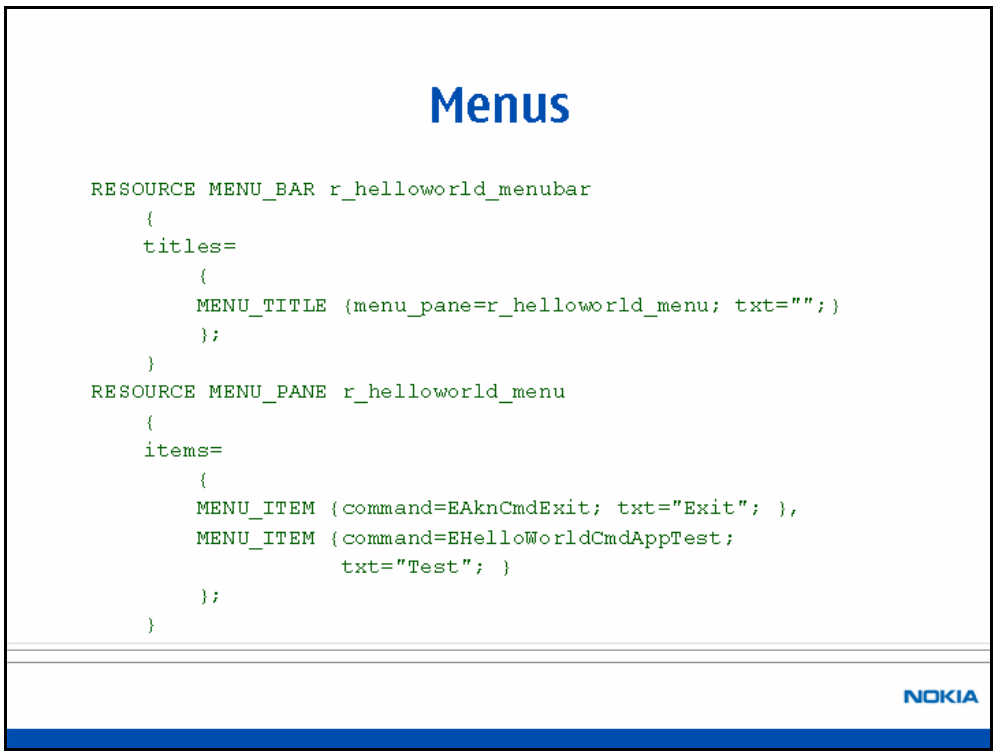
```
{
menubar=r_s60helloworld_menubar;
cba=R_AVKON_SOFTKEYS_OPTIONS_BACK;
}
```

The `RSS_SIGNATURE` allows the application to specify version information, but it is optional to include any content – hence the empty brackets.

The `TBUF` allows the application to specify a friendly name for the default document file, however since the application isn't file based and doesn't have a document file, this is left blank.

The `EIK_APP_INFO` resource determines the menu and the soft-keys that are displayed to the user. The term cba stands for command button area, and equates to the soft-keys in S60. `R_AVKON_SOFTKEYS_OPTIONS_BACK` is a S60 defined option that displays an options menu on the left soft key and a back button on the right soft key. The menubar `r_helloworld_menubar` specifies the menu options and is discussed on the next slide.

Menus



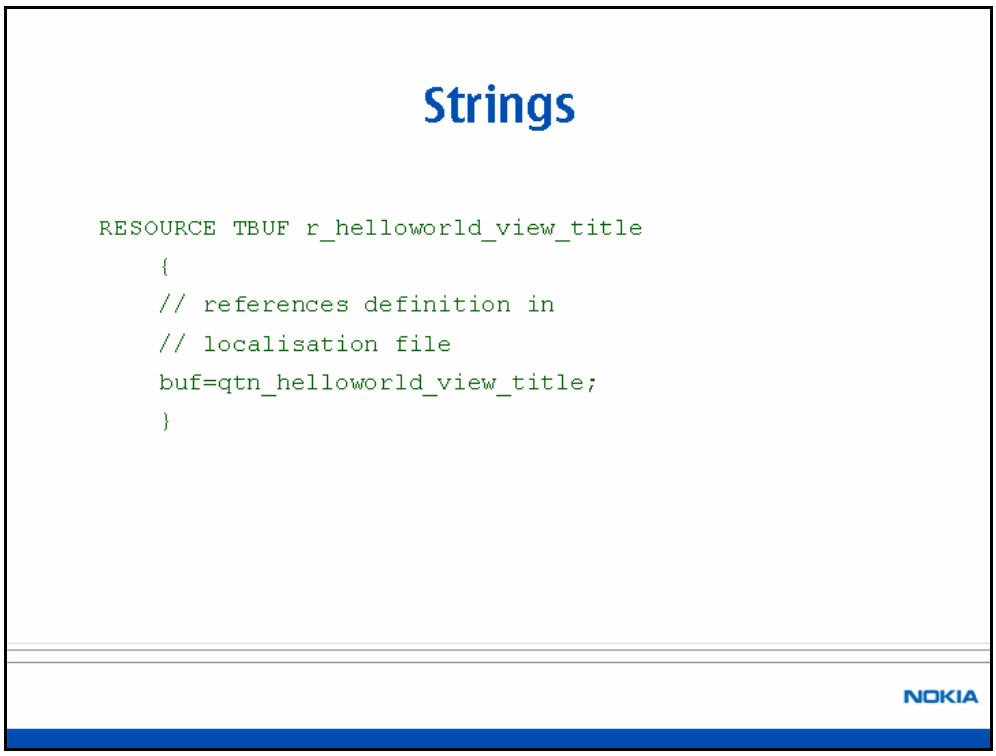

```
RESOURCE MENU_BAR r_helloworld_menubar
    {
    titles=
        {
        MENU_TITLE { menu_pane=r_helloworld_menu; txt=""; }
        };
    }
```

The above menu bar definition defines the menu pane to be displayed. The menu pane contains the menu item definitions. Note that the txt field is blank – this is because its use is not relevant in S60. There is no room to display the title of the menu pane (unlike with Series 80) anywhere so it may be left blank.

```
RESOURCE MENU_PANE r_helloworldapp_menu
    {
    items=
        {
        MENU_ITEM { command=EAknCmdExit; txt="Exit"; },
        MENU_ITEM { command=EHelloWorldCmdAppTest;
            txt="Test"; },
        };
    }
```

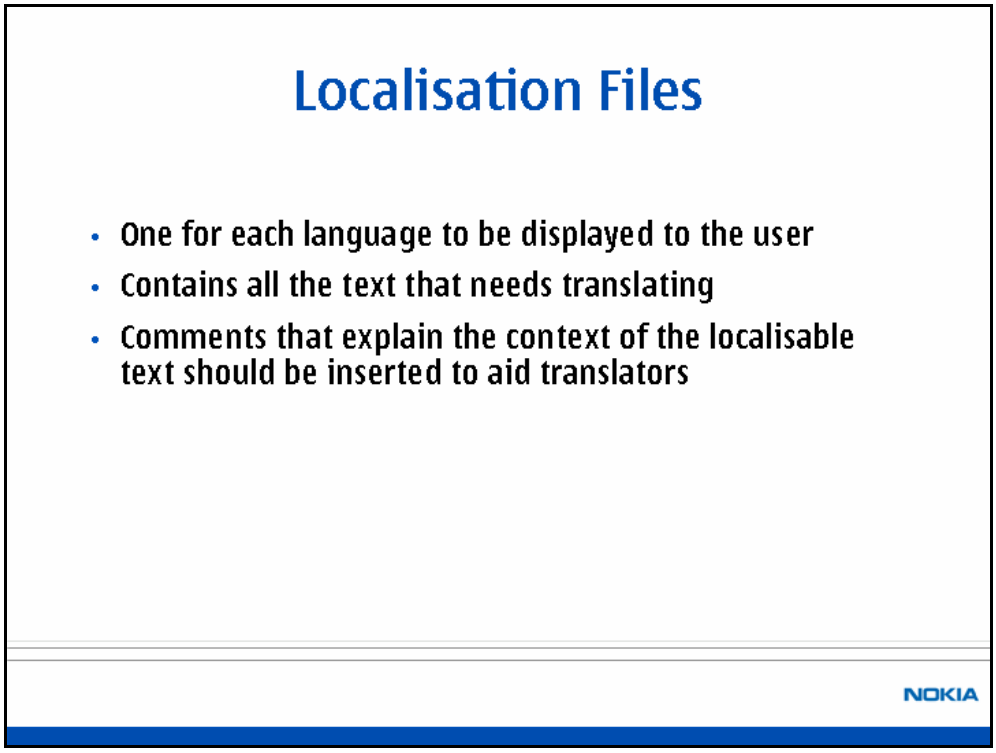
The menu pane defines the menu items available to the application. The command identifiers for each menu item are specified. These are the values that get passed to `CHelloWorldAppUi::HandleCommandL()`. The text for each menu item specifies the text that is presented to the user. Note that this should be defined as a string in the localisation file and referenced here.

Strings



The slide shows a string definition that references a string defined in a localisation file. Localisation files will be covered in the following slides.

Localisation Files

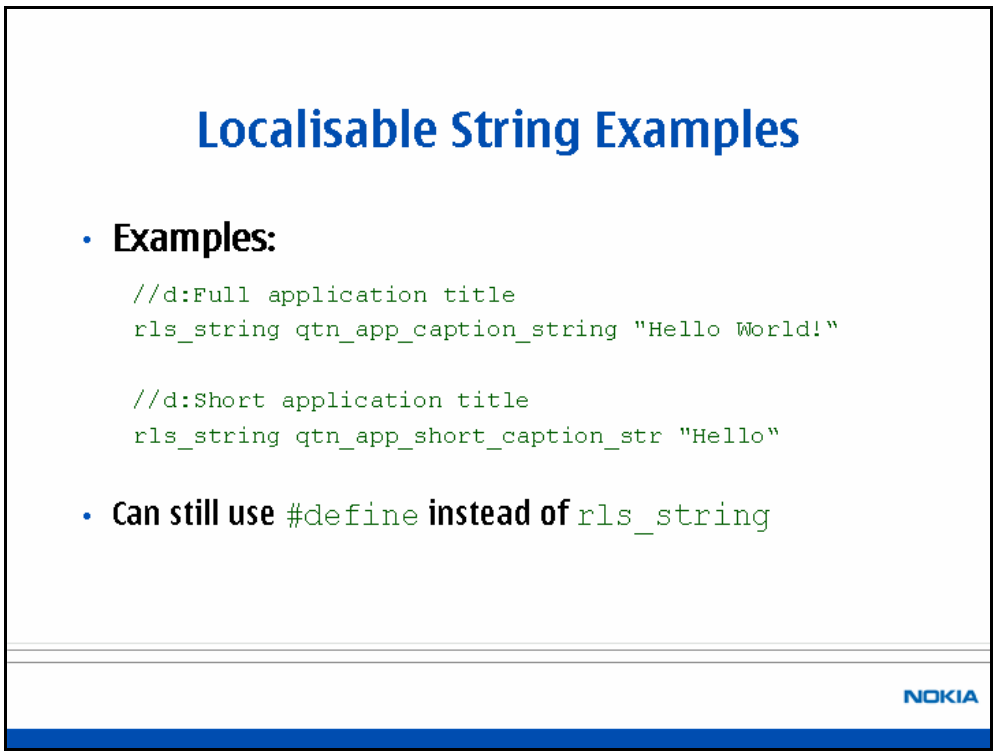


As has been previously mentioned, a localisation file contains all the text that is displayed to the user, which needs to be translated into a different user languages.

During development, all text that is displayed to the user, as part of the application, is defined in a localisation file. When it comes to localising the application, these strings can be translated. Language specific localisation files can then be created. These can be conditionally included in the .rss file. For example:

```
#ifdef LANGUAGE_01
#include "HelloWorld01.rls"
#elif defined LANGUAGE_02
#include "HelloWorld02.rls"
#endif
```

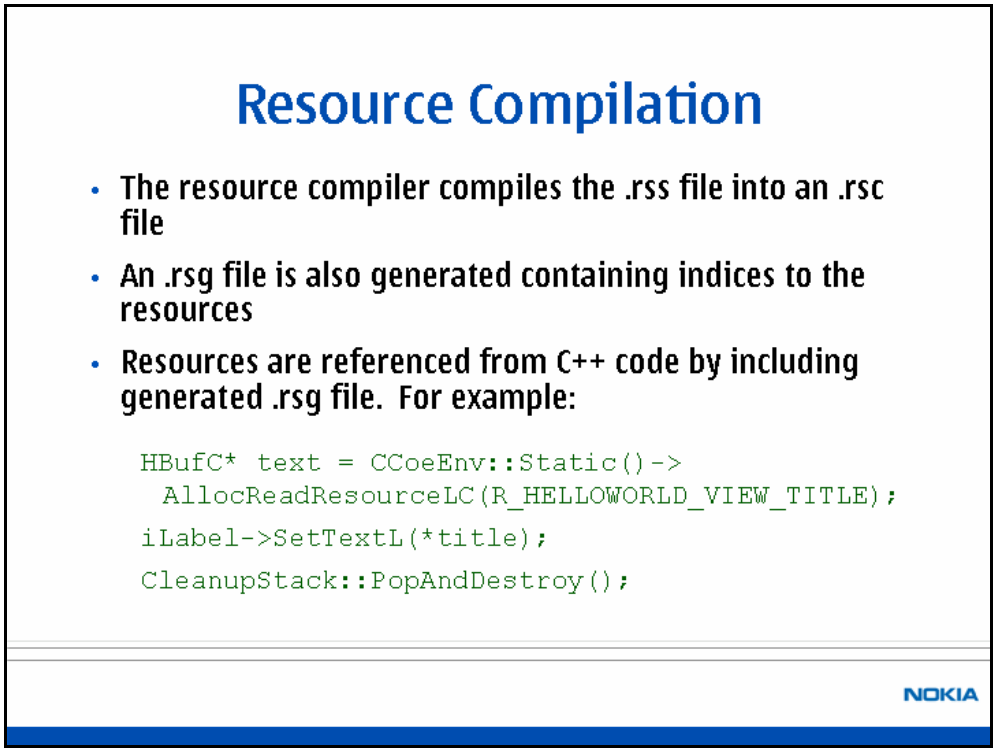
Localisable String Examples



On this slide you see examples of localisable string definitions. A string is specified using the `rls_string` keyword, which is followed by a symbolic identifier and the string itself. The symbolic identifier is used in the resource file for referencing the string. Note the comments above each string definition – these are context information for the translators. The d: comments refer to the usage of the text.

Note that in versions of Symbian OS before v9.x, `#define` was used instead of the keyword `rls_string`. This way of defining localisable strings will still work for Symbian OS v9.x projects (in that your resource files will still compile) but using the `rls_string` keyword is the recommended option.

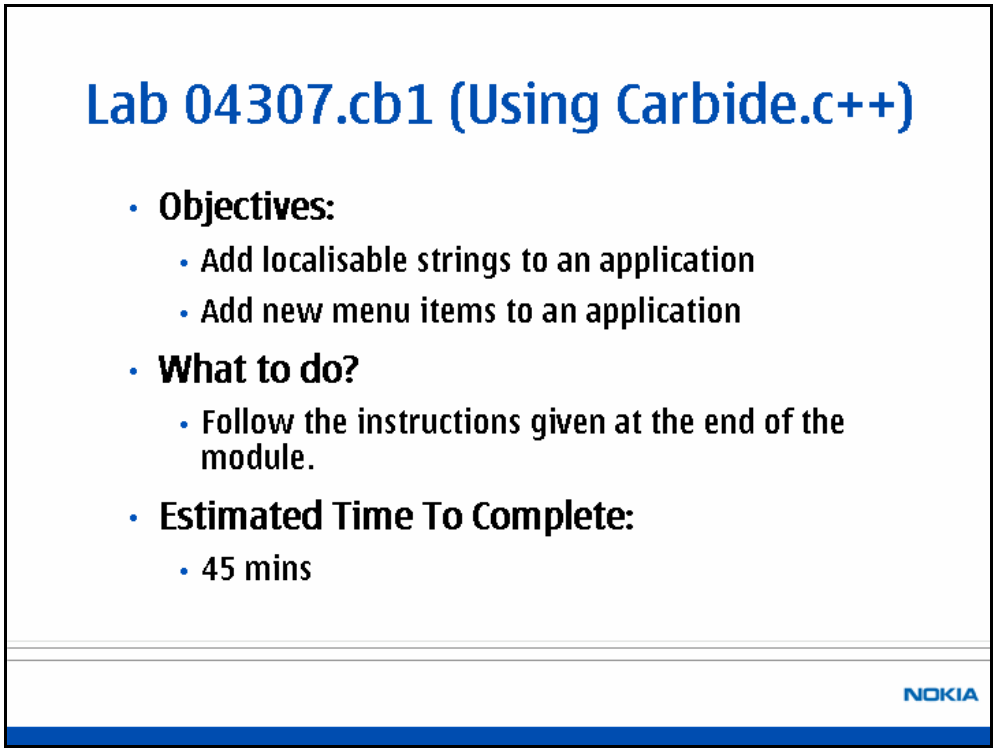
Resource Compilation



At compilation time, the resource file is compiled to produce HelloWorld.rsc. This reduces its size on the target platform. Additionally, HelloWorld.rsg is generated. This gives each resource an index value, so it can be accessed from the C++ code. The identifier of each resource remains the same except that it becomes uppercase. For example, r_HelloWorld_view_title has an index value `R_HELLOWORLD_VIEW_TITLE` generated in the .rsg file.

If localisation is needed, the different languages required should be added to the `LANG` statement within the project .MMP file. This will ensure that separate resource compilation is done for each language conditional compilation.

Lab 04307.cb1 (Using Carbide.c++)



Lab 04307.cb1

Overview

Title:	Using Localisable Strings and Implementing a New Menu Item		
Overview:	This lab shows how to use localisable resource strings.		
Objectives:	To understand how to: • Add localisable strings to an application. • Load string from resource at run time. • Add a menu item to an application. • Handle the menu item selection.		
Compatible IDE(s):	Carbide.c++ v1.1		
Compatible SDK(s):	S60 3rd Edition, S60 3rd Edition (MR), S60 3rd Edition FP1		
App. Type:	Standard GUI	**App. Name:**	S60ResourseLab.exe
Starter Code Provided:	Yes	**Solution Code Provided:**	Yes
Estimated Time To Complete:	45 minutes		

Lab Instructions

Exercise 1 – Preliminary Steps

Task 1.1 – Checks

1. Ensure that a compatible SDK has been installed. A list of compatible SDKs to use with this lab is given in the table in the Introduction.
2. Ensure that a compatible IDE has been installed. A list of compatible IDEs to use with this lab is given in the table in the Introduction.
3. Ensure that the lab files are located in the folder `C:\Labs\Lab_04307.cb1\`. If this is not the case please refer to the setup guide for details of how to obtain them.

Task 1.2 – Make sure the emulator is not in text shell mode

1. Open Windows Explorer.
2. Navigate to the file named epoc.ini, located in the `<EPOCROOT>\epoc32\data` folder. Here, `<EPOCROOT>` is a placeholder for the root folder location of your SDK.
3. Open the epoc.ini file using Notepad.
4. Check that the file does not contain the “textshell” statement. If the statement is present in the file, then either delete the entire line containing the statement or comment the statement out. The latter is done by inserting a # symbol at the start of the line containing the statement.
5. Press ‘Ctrl’ + ‘S’ to save the contents of the file (if you altered it) and close Notepad.

6. Close Windows Explorer.

Task 1.3 – Carbide Setup

1. Launch the Carbide.c++ IDE. Depending on your Carbide.c++ settings the IDE will either display the "Workspace Launcher" dialog or open the last used workspace. If the latter occurs open the "Workspace Launcher" dialog by selecting the "File -> Switch Workspace..." menu item.
2. In the workspace launcher window, create a new workspace by typing `C:\Labs\<Workspace>` in the Workspace field. Here `<Workspace>` is a placeholder for a folder name of you choosing that does not already exist and does not contain spaces.
3. Click the OK button.
4. Close the Welcome window after it appears when the workspace is opened.

Task 1.4 – Importing the project

22. In the Carbide.c++ IDE, select the "File > Import" menu item. The Import dialog appears.
23. Select "Symbian OS bld.inf file" from the "Symbian OS" subfolder as the import source and click the Next button. A new screen appears on the Import dialog.
24. Enter `C:\Labs\Lab_04307.cb1\starter\group\bld.inf` in the "bld.inf file" field. Alternatively, click the "Browse..." button and then navigate to and select the correct file via the dialog. When this is done a list of configurations appears in the "SDKs and Build Configurations pane".
25. Expand the list of configurations for your installed SDK and tick the checkbox next to the "Emulator Debug" configuration. (Note that other configurations may be chosen in addition to this one, but they are not used in this lab.) Then select the "Next" button.
26. Ensure the MMP file and all the make files are selected on the "MMP Selection" screen. Then select the Next button. The Project Properties screen appears.
27. Change the name in the Project field to "S60ResourseLab" (without the quotes) and click the Finish button to complete the import.
28. The Import dialog closes and a new project called "S60ResourseLab" appears in the "C/C++ Projects" view.

1. If not already present, enter `S60ResourseLab` in the "Into folder" field and select the Finish button. A new `gfx` folder should appear in your project containing the necessary graphics files.

Exercise 2 – How to use localisable resource strings

Task 2.1 – Adding localisable resource strings

1. Double click on the `\S60ResourceLab\data\S60ResourceLab.rls` file node in "C/C++ Projects" view. A text editor window is opened showing the contents of the file.
2. Add additional localisable resource strings to the file as shown by the code in bold below:

```
// Copyright (c) 2006 Nokia Corporation.

// LOCALISATION STRINGS

rls_string qtn_caption_string "S60 Resource Lab"
rls_string qtn_s60resourcelab_text_goodbye "Goodbye"
```

```
rls_string qtn_s60resourcelab_text_everyone "Everyone"
rls_string qtn_s60resourcelab_menu_goodbye "Goodbye"
rls_string qtn_loc_resource_file_1
"\\resource\\apps\\S60ResourceLab"
rls_string qtn_s60resourcelab_menu_exit "Exit"
rls_string qtn_s60resourcelab_menu_test "Test"
rls_string qtn_s60resourcelab_text_hello_world "Hello World!"
rls_string qtn_s60resourcelab_text_hello "Hello"
rls_string qtn_s60resourcelab_text_world "World"

// End of File
```

3. Press 'Ctrl' + 'S' to save the contents of the file.
4. Close the text editor window for `S60ResourceLab.rls`.

Task 2.2 – Using the localisable resource strings in resources

1. Double click on the `\S60ResourceLab\Data\S60ResourceLab.rss` file node in "C/C++ Projects" view. A text editor window is opened showing the contents of the file.
2. Edit the `r_s60resourcelab_menu MENU_PANE` resource, as illustrated by the code in bold below, so it references localisable resource strings instead of using hard coded strings:

```
RESOURCE MENU_PANE r_s60resourcelab_menu
    {
    items=
        {
        MENU_ITEM { command=EAknCmdExit;
                    txt=qtn_s60resourcelab_menu_exit; },
        MENU_ITEM { command=ES60ResourceLabCmdAppTest;
                    txt=qtn_s60resourcelab_menu_test; }
        };
    }
```

3. Add new `TBUF` resources, as shown by the code in bold below, that reference the non-menu localisable resource strings you added earlier:

```
...
RESOURCE TBUF r_s60resourcelab_text_everyone
    {
    buf = qtn_s60resourcelab_text_everyone;
    }

RESOURCE TBUF r_s60resourcelab_text_hello_world
    {
    buf = qtn_s60resourcelab_text_hello_world;
    }

RESOURCE TBUF r_s60resourcelab_text_hello
    {
    buf = qtn_s60resourcelab_text_hello;
    }

RESOURCE TBUF r_s60resourcelab_text_world
    {
    buf = qtn_s60resourcelab_text_world;
```

```
    }
...
```

4. Press 'Ctrl' + 'S' to save the contents of the file.
5. Close the text editor window for `S60ResourceLab.rss`.

Task 2.3 – Replacing hard-coded strings with localisable resource strings

1. Double click on the `\S60ResourceLab\src\S60ResourceLabAppUi.cpp` file node in "C/C++ Projects" view. A text editor window is opened showing the contents of the file.
2. As illustrated by the code in bold below, edit the `HandleCommandL` function so that the code executed for the `ES60ResourceLabCmdAppTest` command loads a localisable string from the resources instead of using a hard-coded string.

```
void CS60ResourceLabAppUi::HandleCommandL(TInt aCommand)
    {
    switch ( aCommand )
        {
        case EAknSoftkeyBack:
        case EEikCmdExit:
            {
            Exit();
            break;
            }

        case ES60ResourceLabCmdAppTest:
            {
            HBufC* helloworld = iCoeEnv-
>AllocReadResourceL(R_S60RESOURCELAB_TEXT_HELLO_WORLD);
            iEikonEnv->InfoMsg(*helloworld);
            delete helloworld;
            break;
            }

        default:
            break;
        }
    }
```

3. Press 'Ctrl' + 'S' to save the contents of the file.
4. Close the text editor window for `S60ResourceLabAppUi.cpp`.
5. Double click on the `\S60ResourceLab\src\S60ResourceLabContainer.cpp` file node in "C/C++ Projects" view. . A text editor window is opened showing the contents of the file.
6. As shown by the code in bold below, edit the `ConstructL` function so that it loads localisable strings from the resources instead of using a hard-coded strings.

```
void CS60ResourceLabContainer::ConstructL(const TRect& aRect)
    {
    CreateWindowL();

    iLabel = new (ELeave) CEikLabel;
    iLabel->SetContainerWindowL( *this );
```

```
    HBufC* hello = iCoeEnv-
>AllocReadResourceLC(R_S60RESOURCELAB_TEXT_HELLO);
    iLabel->SetTextL(*hello);

    iToDoLabel = new (ELeave) CEikLabel;
    iToDoLabel->SetContainerWindowL( *this );

    HBufC* world = iCoeEnv-
>AllocReadResourceLC(R_S60RESOURCELAB_TEXT_WORLD);
    iToDoLabel->SetTextL(*world);

    CleanupStack::PopAndDestroy(2);

    SetRect(aRect);
    ActivateL();
    }
```

Note that the use of the `AllocReadResourceLC` function means that our descriptors are added to the cleanup stack. The function `CleanupStack::PopAndDestroy` is used to remove them once they are no longer required.

7. Press 'Ctrl' + 'S' to save the contents of the file.
8. Close the text editor window for `S60ResourceLabContainer.cpp`.

Task 2.4 – Building and running the application

1. Build the application for the Emulator Debug configuration of the SDK you are using by right clicking on the S60ResourceLab project node in "C/C++ Projects" view and select the "Build Project" or "Rebuild Project" menu item.
2. Right click on the S60ResourceLab project node in "C/C++ Projects" view and select the "Run As > Run Symbian OS Application" menu item. The emulator will launch.
3. Follow the instructions for this step that are specific to the SDK you are using:

   S60 3rd Edition SDK and S60 3rd Edition MR SDK

   After the emulator has booted the Applications menu is shown. The S60ResourseLab application is located in the "Installat." folder. Navigate to this folder using the navigation keys and open it using the selection key.

   S60 3rd Edition FP1 SDK

   After the emulator has booted the Standby application is shown. Switch to the Applications menu by selecting the Applications key. The S60ResourseLab application is located in the "Installed" folder. Navigate to this folder using the navigation keys and open it using the selection key.
4. The S60ResourceLab application is located in the "Installat." folder. Navigate to this folder using the navigation keys and open it using the selection key.
5. Navigate to the S60ResourceLab application, again using the navigation keys, and launch it using the selection key. The application is launched and displays two labels on the screen. The top label contains the text "Hello" and the bottom label contains the text "World".
6. Click the Options soft-key (or press F1 on the PC keyboard). The Options menu appears.
7. Select the Test menu item (using the navigation keys and the select softkey). A message is displayed in the top right corner of the screen containing the text "Hello World!". The message disappears after a few seconds.
8. Exit the application by selecting the Back soft-key.

9. Close the emulator.

Exercise 3 – Adding and using a new menu item

Task 3.1 – Adding a new command ID

1. Double click on the `\S60ResourceLab\inc\S60resourcelab.hrh` file node in "C/C++ Projects" view. A text editor window is opened showing the contents of the file.
2. As shown by the code in bold below, add an additional command ID for a new 'Goodbye' menu item:

```
enum TS60ResourceLabCommandIds
    {
    ES60ResourceLabCmdAppTest = 1,
    ES60ResourceLabCmdGoodbye
    };
```

   Do not forget to add a comma after the existing command ID.
3. Press 'Ctrl' + 'S' to save the contents of the file.
4. Close the text editor window for `S60ResourceLab.hrh`.

Task 3.2 – Adding a new menu item

1. Double click on the `\S60ResourceLab\data\S60ResourceLab.rss` file node in "C/C++ Projects" view. A text editor window is opened showing the contents of the file.
2. As shown by the code in bold below, add a 'Goodbye' menu item to the `r_s60resourcelab_menu MENU_PANE` resource:

```
RESOURCE MENU_PANE r_s60resourcelab_menu
    {
    items=
        {
        MENU_ITEM { command=EAknCmdExit;
                    txt=qtn_s60resourcelab_menu_exit; },
        MENU_ITEM { command=ES60ResourceLabCmdAppTest;
                    txt=qtn_s60resourcelab_menu_test; },
        MENU_ITEM { command=ES60ResourceLabCmdGoodbye;
                    txt=qtn_s60resourcelab_menu_goodbye; }
        };
    }
```

   Do not forget to add a comma after the 'Test' menu item.

   Note that the localisable resource string, `qtn_s60resourcelab_menu_goodbye`, is already defined in `S60ResourceLab.rls`.
3. Press 'Ctrl' + 'S' to save the contents of the file.
4. Close the text editor window for `S60ResourceLab.rss`.

Task 3.3 – Adding handler code for the new menu item

1. Double click on the `\S60ResourceLab\src\S60ResourceLabAppUi.cpp` file node in "C/C++ Projects" view. A text editor window is opened showing the contents of the file.
2. As shown by the code in bold below, add handler code to the `HandleCommandL` function

for the Goodbye menu item:

```
void CS60ResourceLabAppUi::HandleCommandL(TInt aCommand)
    {
    switch ( aCommand )
        {
        case EAknSoftkeyBack:
        case EEikCmdExit:
            {
            Exit();
            break;
            }

        case ES60ResourceLabCmdAppTest:
            {
            HBufC* hello = iCoeEnv-
>AllocReadResourceL(R_S60RESOURCELAB_TEXT_HELLO_WORLD);
            iEikonEnv->InfoMsg(*hello);
            break;
            }

        case ES60ResourceLabCmdGoodbye:
            {
            iAppContainer->GoodbyeL();
            break;
            }

        default:
            break;
        }
    }
```

The handler code calls the `CS60ResourceLabContainer::GoodbyeL` function which replaces the text in the labels with “Goodbye” and “Everyone”.

3. Press ‘Ctrl’ + ‘S’ to save the contents of the file.
4. Close the text editor window for `S60ResourceLabAppUi.cpp`.

Task 3.4 – Building and running the application

1. As described in the previous exercise, rebuild the application, run the emulator, navigate to and launch the S60ResourceLab application.
2. Click the Options soft-key and select the Goodbye menu item. The text in the two labels on the emulator screen changes from “Hello” and “World” to “Goodbye” and “Everyone”.
3. Exit the application and close the emulator.
4. Close the Carbide.c++ IDE. This completes the lab.

Module #04308

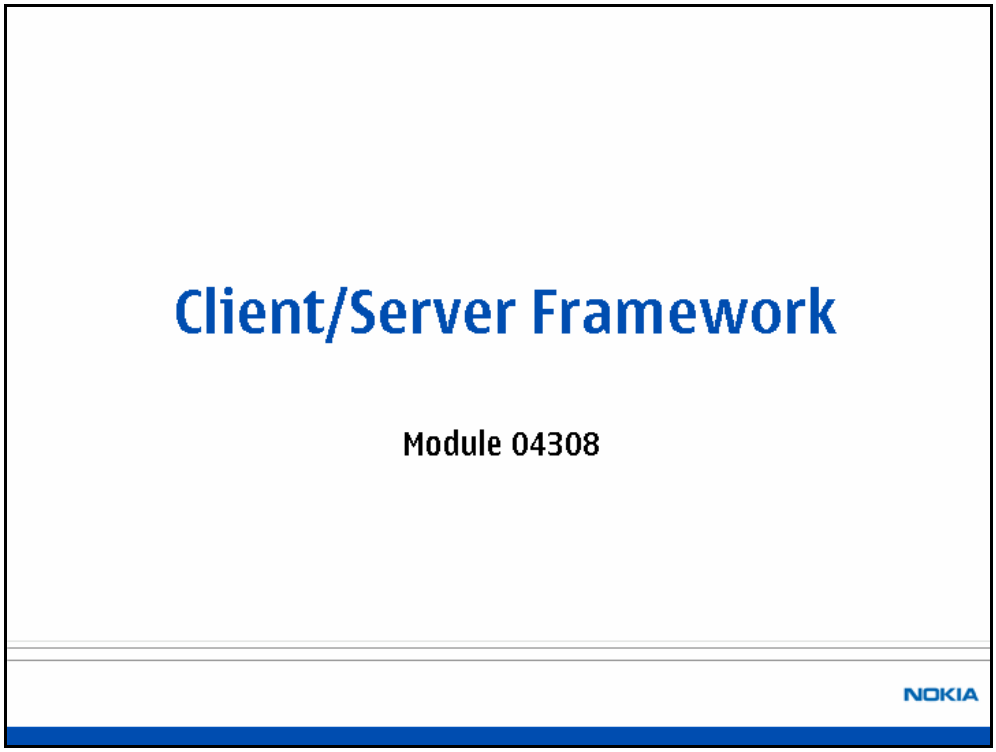
Client/Server Framework

Contents

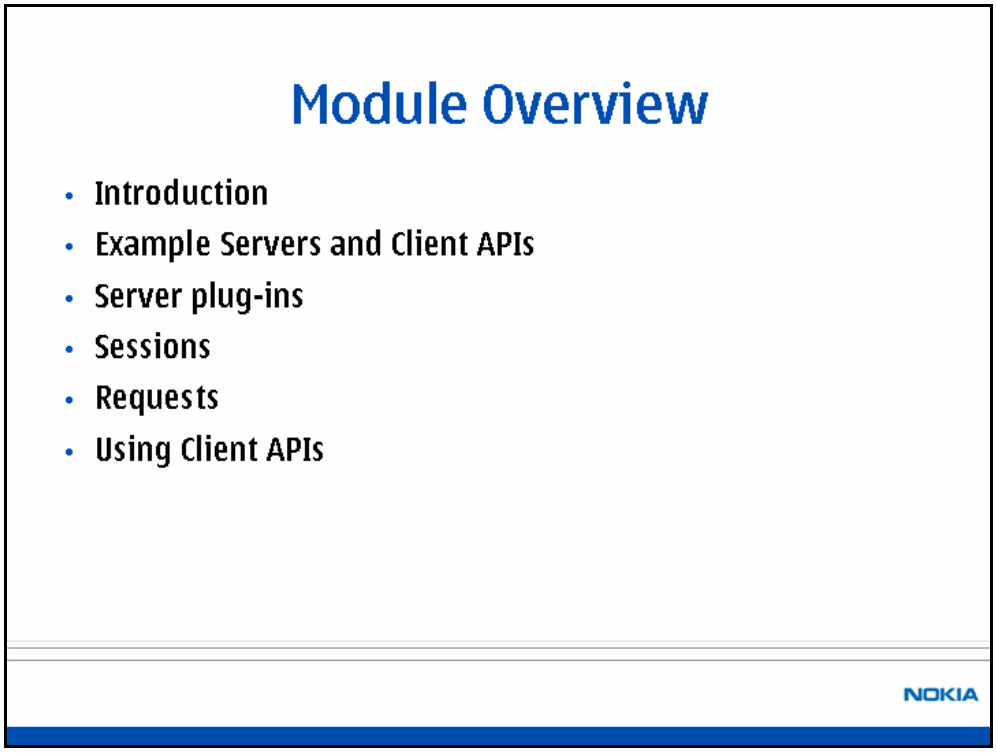
NOKIA

BO-EN-04300-20080108

Client/Server Framework

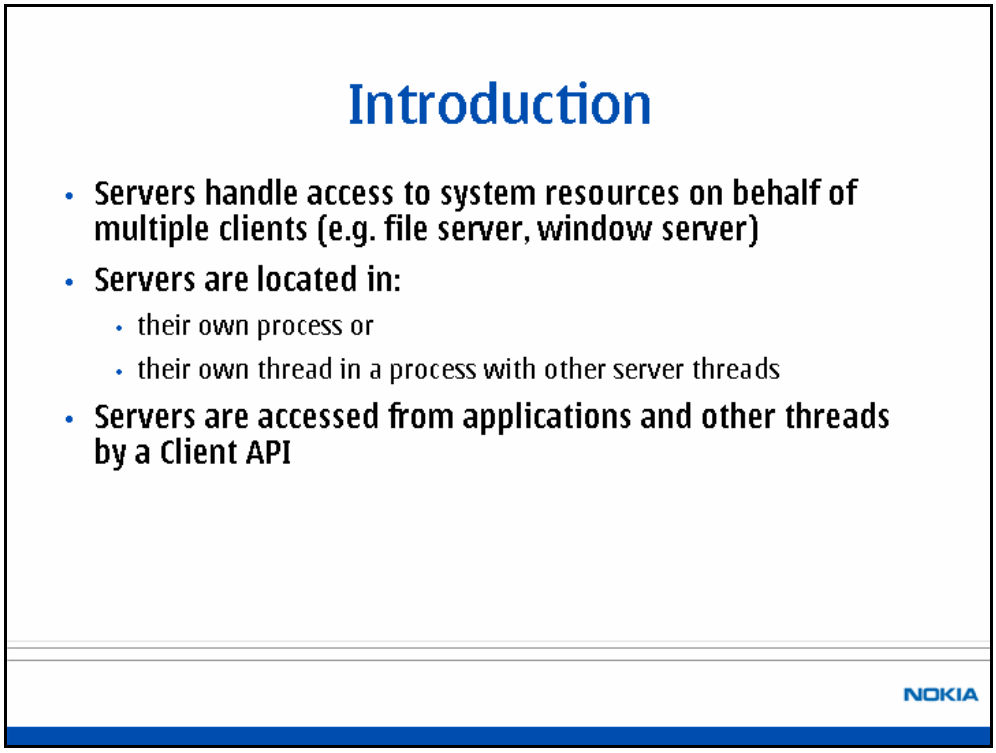


Module Overview



This lesson looks at the client/server framework from a users perspective. It focuses on developers using the client API rather than implementing server functionality.

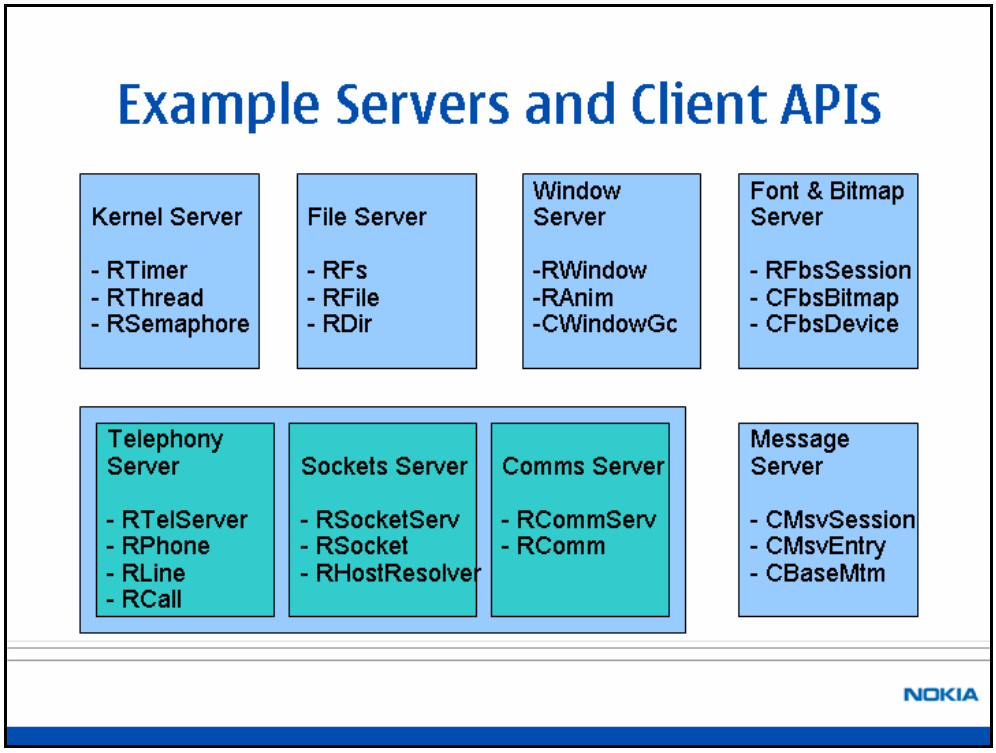
Introduction



The client/server framework is used to access system resources such as opening files, making telephone calls, setting the alarm, and so on. These resources need to be carefully controlled so that concurrent access from different threads will not cause any problems. It is the server that handles the access to the resource and the client that requests access.

Servers run within their own process or thread and any requests from clients are done across thread boundaries. This means that the server has the ultimate control over the resource and can fail any requests that would cause an error, for example a file delete request from one application, when another application was editing the file.

Example Servers and Client APIs



The slide illustrates some of the main servers on a Symbian OS device. Each of the boxes represents a different server:

- The blue ones are located in their own process.
- The green ones are located in a separate thread in a process shared with other servers.

The classes listed within each server represent APIs that access the server from client threads. Note that most of these begin with the letter 'R'. Some of them, however, are 'C' classes. Much of the functionality in these servers is covered by other modules. The following is a brief summary of the servers and client APIs shown on the slide.

Kernel server - This runs with supervisor privilege and controls access to hardware and memory from all other processes on the system. Examples of client access to the kernel are as follows:

- `RTimer` - Provides asynchronous timer services.
- `RThread` - Provides thread access and creation.
- `RSemaphore` - Allows synchronisation between threads.

File server - This provides access to the filing system. The API allows file and directory creation, renaming and deletion in addition to file reading and writing. Examples of file server client APIs are as follows:

- `RFs` - Provides a session to the file server. High level drive, directory and file operations can be done here in addition to getting directory listings.
- `RFile` - Allows file creation, reading and writing.

- `RDir` - Reads entries contained in a directory.

Window Server - The application framework uses the window server to handle key events and drawing to the screen. The following are example APIs:

- `RWindow` - This is used to enable drawing to the screen. It is not commonly used by the application developer. Instead drawing is normally done in the `CCoeControl::Draw()` override.
- `RAnim` - This is used to communicate with a server side animation (high priority drawing in a thread of the window server).
- `CWindowGc` - Graphics context providing drawing functions to draw to the screen.

Font and Bitmap server - This shares fonts and bitmaps between clients. The following are example client APIs:

- `RFbsSession` - Is responsible for a session with the font and bitmap server. It is not normally used in application development.
- `CFbsBitmap` - This represents a bitmap.
- `CFbsDevice` - This represents a graphics device that is used to draw to a bitmap.

Telephony server - This is responsible for all device telephony. Example client APIs are as follows:

- `RTelServer` - Provides root level access to the telephony server. TSYs (telephony extension modules) can be loaded and queries as to the phones on a device can be made.
- `RPhone` - Provides operations for a given phone on a device.
- `RLine` - Provides functionality for a line with a phone.
- `RCall` - Provides functionality to make/receive a call over a line.

Sockets server - Provides functionality to make socket TCP/IP and UDP socket connections across a variety of media. The following APIs are examples of socket server functionality:

- `RSocketServ` - Connects to the socket server and can discover the available protocols.
- `RSocket` - Provides functionality to connect, receive and send data to another socket.
- `RHostResolver` - Performs DNS resolution and gets the addresses of remote devices.

Comms server - Allows developers to use serial ports via cable or infra-red.

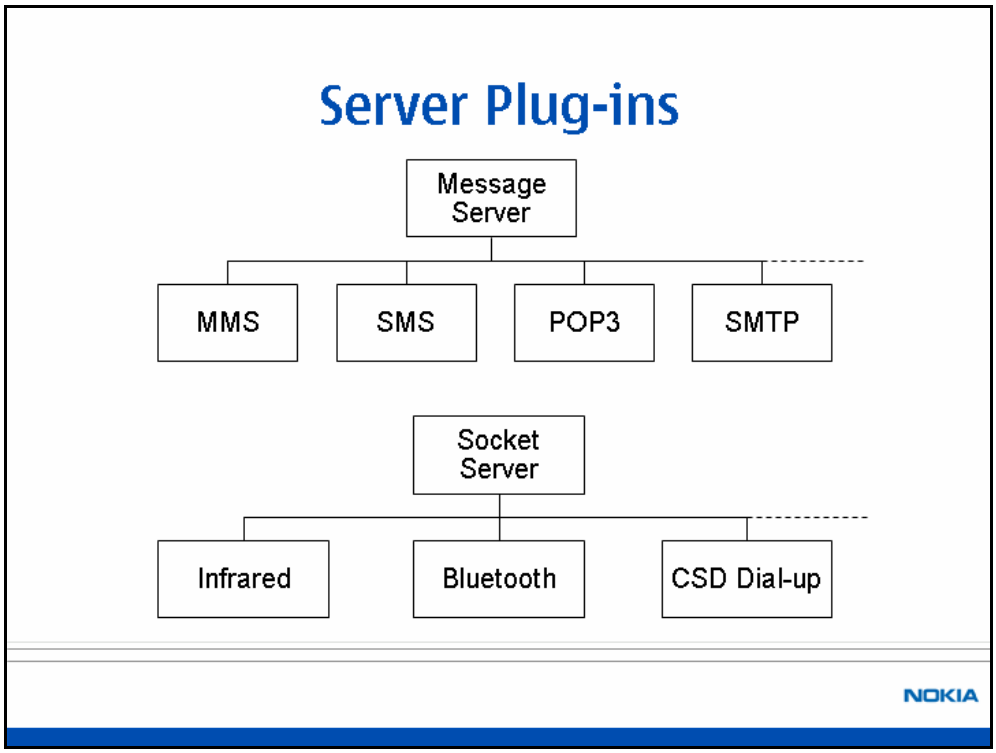
- `RCommServ` - Provides a session with the comms server.
- `RComm` - Provides the necessary functions for communicating via a serial port.

Message Server - This stores message data and provides access to messaging functionality for MMS, SMS, OBEX and email. The following example APIs access the message server:

- `CMsvSession` - Represents a session with the message server. It provides functionality to access the message store and be notified of message server related events.

- `CMsvEntry` - Represents an entry in the message store.
- `CBaseMtm` - Provides a high-level interface for accessing and manipulating a message server entry.

Server Plug-ins



The client/server architecture has a fair degree of extensibility to it. Some servers allow plug-in modules to be added to them to offer functionality for new technologies, protocols and media. This additional functionality is accessed via the generic API as normal, but may use a different initialisation constant or parameter. The slide illustrates two different examples of this plug-in architecture:

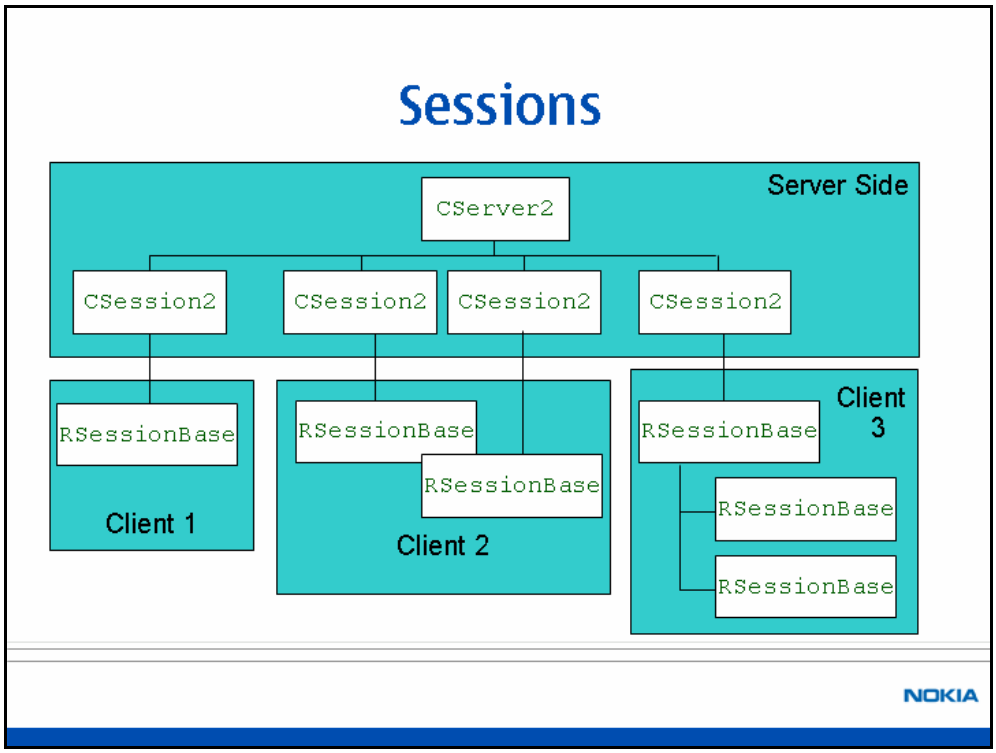
The message server allows messages to be sent and received via a range of messaging protocols. It allows various Message Type Modules (MTMs) to be added. An MTM encapsulates the functionality required to send and/or receive a specific type of message:

- The MMS MTM provides an API to send multimedia messages.
- The SMS MTM provides an API to send text messages.
- The POP3 MTM provides an API to receive email.
- The SMTP MTM provides an API to send email.

The socket server allows TCP/IP and UDP functionality to be used across a range of different media depending on the protocol constants being used in the `RSocketServ` and `RSocket` APIs. The following are examples of the different media allowed:

- Infra-red socket communications.
- Bluetooth socket communications.
- Circuit switched data socket communications via a dial-up connection to an ISP.

Sessions



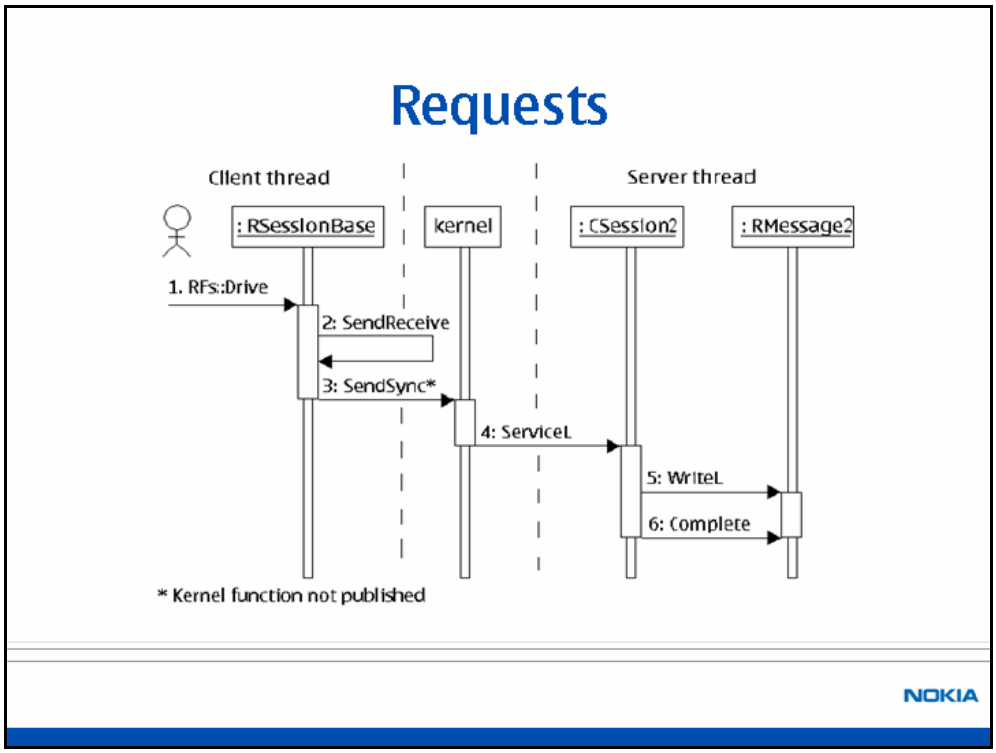
The slide shows different client / server session scenarios. A server can support sessions from multiple client threads and additionally multiple sessions from the same client thread. The green boxes indicate the thread boundaries. The more sessions, the more system resources are required.

- Client 1 has a single session with the server. An example of this would be the default session with the file server within the control environment of an application.
- Client 2 has two sessions with a server. This uses up more resources than having a single session, even though both sessions are from the same thread. An example of using two server sessions within an application is having two sessions with the file server - the default one from the control environment (`CCoeEnv`), and the other opened explicitly to perform another file related operation.
- Client 3 has one session with a server and two subsessions. Subsessions are a less resource costly way to have two logical connections with a server. An example of this scenario is a client having one connection to the file server and a separate subsession for each file opened. The client file handle, `RFile`, derives (indirectly) from `RSubSessionBase`.

The classes shown in the slide are all base classes of what is actually implemented. A brief explanation of each is given below:

- `CServer2` – Base class for all servers. Handles connections from clients.
- `CSession2` – Handles requests from the client once connection has been established (an example request would be renaming a file in the file server).
- `RSessionBase` – Allows clients to connect to the server and make requests.
- `RSubSessionBase` – Provides client subsession access to the server.

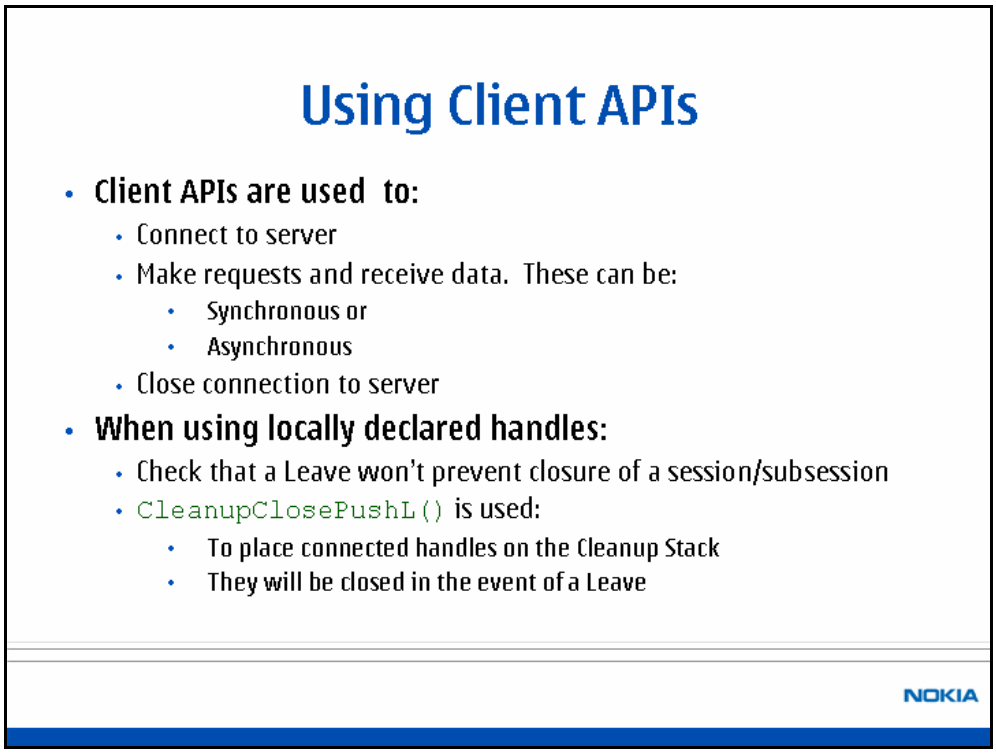
Requests



The slide shows how clients communicate across thread boundaries with a server. Note that the classes `RSessionBase`, `CSession2` and `RMessage2` are base classes and derivations are needed. The following points explain the process in more detail:

- An API offered by the client interface is called, for example `RFs::Drive()` in the file server client API to get drive information.
- `RSessionBase::SendReceive()` is called. The operation code is passed as a parameter to this function. In addition, an array of 4 pointers to client memory is also passed which allow the server to get and set client data.
- Internally the private function `RSessionBase::SendSync()` is called. This calls the kernel.
- The kernel calls `CSession2::ServiceL()` in the server thread. A reference to the message contents, encapsulated by `RMessage2`, is passed as a parameter. The operation code is evaluated to see what action is taken.
- `RMessage2::WriteL()` is called to set data in the client address space.
- `RMessage2::Complete()` is called to signal completion of a client request.

Using Client APIs

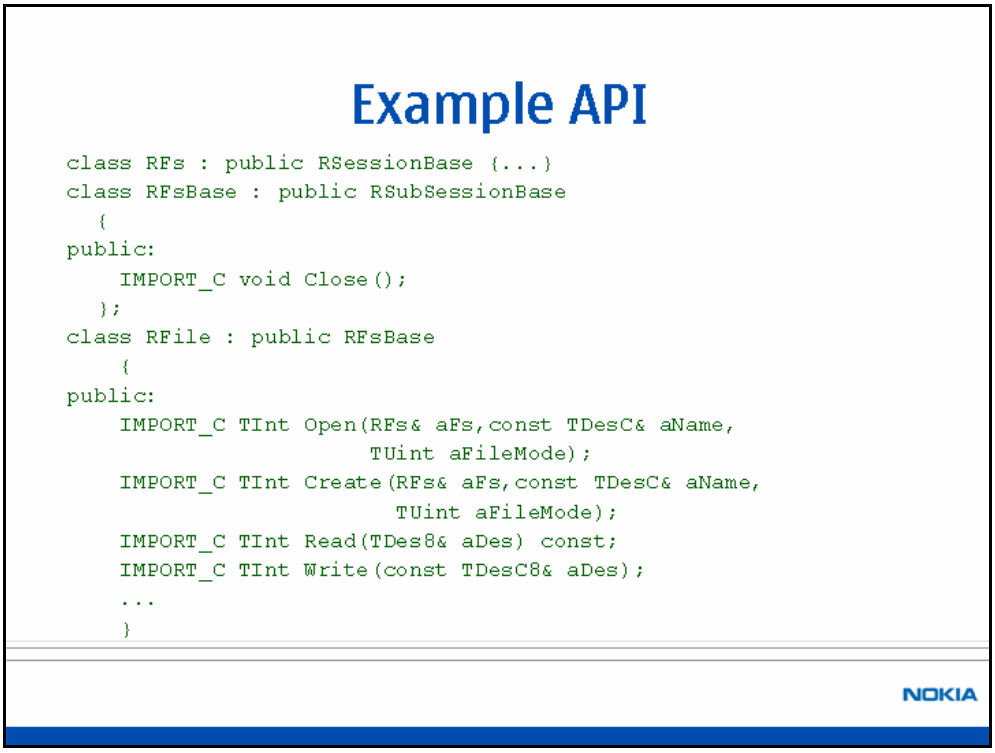


All client APIs are derived from either `RSessionBase` or `RSubSessionBase`. They tend to be used in the following order:

- A connection must first be made with the associated server, from the `RSessionBase` derived class. Subsessions are usually created by `Connect()` or `Open()` APIs that take a handle to an already connected session.
- The main APIs can be used after connection. These will obviously vary widely depending on the type of client. Requests can either be made synchronously or asynchronously. The Active Object Framework lesson that follows this, will illustrate how to handle asynchronous requests.
- The connection to a server must be closed after use otherwise a resource leak will occur.

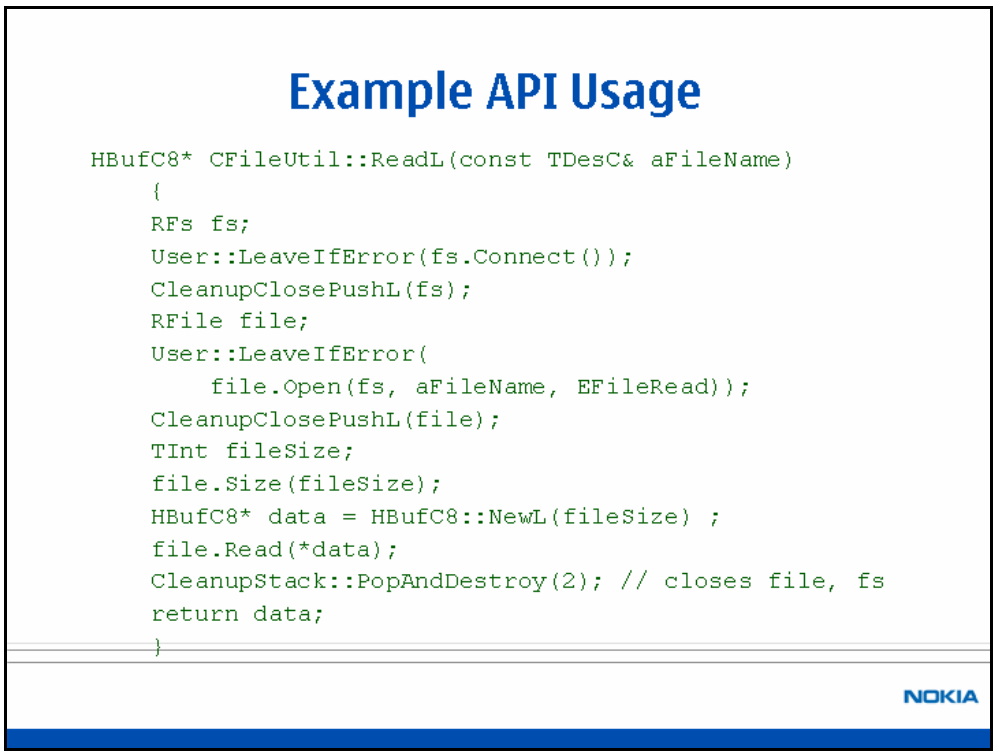
Care should be taken when using locally declared handles as a leave will cause the handle to be left dangling. To get around this problem, handles that need closing can be put on the cleanup stack using the global function `CleanupClosePushL()`. Any handles placed on the cleanup stack using this function will be closed in the event of a leave.

Example API



The slide shows part of the file server client API. Note that the handle to the file server session, `RFs` is derived from `RSessionBase`. The `RFile` API is derived indirectly from `RSubSessionBase`, meaning that multiple files can be opened in a given server session. The `RFile::Open` and `RFile::Create` functions take a parameter that is a reference to `RFs`. The direct parent of `RFile`, `RFsBase` provides a `Close()` function which must be called when no more file access is required. This will clean up the resources associated with the file.

Example API Usage

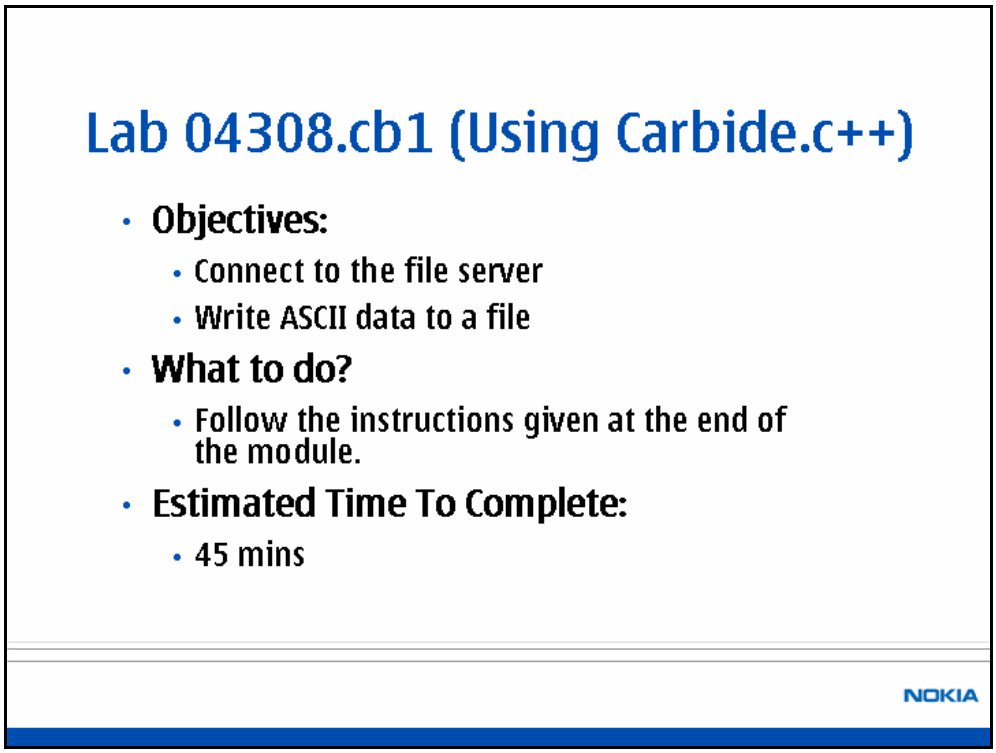


The example on the slide provides example code that illustrates how to read all the data in a file synchronously. Each line of code is explained below:

1. A handle to the file server is declared on the stack.
2. `RFs::Connect()` is called to connect to the file server. This will return an error code indicating whether it has been successful. `User::LeaveIfError()` is used to raise a leave if the error code is anything other than `KErrNone`.
3. If the code execution gets to this line, the connection to the file server has succeeded. The open handle is put on the cleanup stack in case a leave occurs later.
4. A handle to a file is declared on the stack.
5. A file that matches the filename (including path) passed to the function is opened in read-only mode. Note that the already connected handle to the file server is passed as a parameter. If the open fails then a leave is raised.
6. The handle to the now successfully opened file is placed on the cleanup stack.
7. An integer parameter to store the size of the file is declared on the stack.
8. The `fileSize` variable is passed as a parameter to `RFile::Size()` to get the size of the opened file. Note the unconventional naming of the function (should be `GetSize()` when the value returned is an out parameter).
9. A buffer of the size of the file contents is declared on the heap.
10. The file contents are read into the buffer.
11. The last two items on the cleanup stack are closed and removed from the stack.

12. The data is returned to the caller. It is the caller's responsibility to delete it.

Lab 04308.cb1 (Using Carbide.c++)



Lab 04308.cb1

Overview

Title:	Creating a session with the file server.		
Overview:	This lab shows how to use a simple client API to connect to a server and perform some operations. The file server API is one of the simpler ones so will be used in this case. All the APIs used are synchronous in keeping with this simplicity - the active object framework labs will introduce the concept of asynchronous APIs and how to handle them.		
Objectives:	To understand how to: • Open and close a file server session. • Create a file or directory. • Write data to a file. • Free all resources after use.		
Compatible IDE(s):	Carbide.c++ v1.2		
Compatible SDK(s):	S60 3rd Edition, S60 3rd Edition (MR), S60 3rd Edition FP1		
App. Type:	Standard GUI	**App. Name:**	S60ClientServerLab.exe
Starter Code Provided:	Yes	**Solution Code Provided:**	Yes
Estimated Time To Complete:	45 minutes		

Lab Instructions

Exercise 1 – Preliminary Steps

Task 1.1 – Checks

1. Ensure that a compatible SDK has been installed. A list of compatible SDKs to use with this lab is given in the table in the Introduction.
2. Ensure that a compatible IDE has been installed. A list of compatible IDEs to use with this lab is given in the table in the Introduction.
3. Ensure that the lab files are located in the folder `C:\Labs\Lab_04308.cb1\`. If this is not the case please refer to the setup guide for details of how to obtain them.

Task 1.2 – Make sure the emulator is not in text shell mode

1. Open Windows Explorer.
2. Navigate to the file named epoc.ini, located in the `<EPOCROOT>\epoc32\data` folder. Here, `<EPOCROOT>` is a placeholder for the root folder location of your SDK.
3. Open the epoc.ini file using Notepad.

4. Check that the file does not contain the “textshell” statement. If the statement is present in the file, then either delete the entire line containing the statement or comment the statement out. The latter is done by inserting a # symbol at the start of the line containing the statement.
5. Press ‘Ctrl’ + ‘S’ to save the contents of the file (if you altered it) and close Notepad.
6. Close Windows Explorer.

Task 1.3 – Carbide Setup

1. Launch the Carbide.c++ IDE. Depending on your Carbide.c++ settings the IDE will either display the “Workspace Launcher” dialog or open the last used workspace. If the latter occurs open the “Workspace Launcher” dialog by selecting the “File -> Switch Workspace...” menu item.
2. In the workspace launcher window, create a new workspace by typing `C:\Labs\<Workspace>` in the Workspace field. Here `<Workspace>` is a placeholder for a folder name of you choosing that does not already exist and does not contain spaces.
3. Click the OK button.
4. Close the Welcome window after it appears when the workspace is opened.

Task 1.4 – Importing the project

29. In the Carbide.c++ IDE, select the “File > Import” menu item. The Import dialog appears.
30. Select “Symbian OS bld.inf file” from the “Symbian OS” subfolder as the import source and click the Next button. A new screen appears on the Import dialog.
31. Enter `C:\Labs\Lab_04308.cb1\starter\group\bld.inf` in the “bld.inf file” field. Alternatively, click the “Browse...” button and then navigate to and select the correct file via the dialog. When this is done a list of configurations appears in the “SDKs and Build Configurations pane”.
32. Expand the list of configurations for your installed SDK and tick the checkbox next to the “Emulator Debug” configuration. (Note that other configurations may be chosen in addition to this one, but they are not used in this lab.) Then select the “Next” button.
33. Ensure the MMP file and all the make files are selected on the “MMP Selection” screen. Then select the Next button. The Project Properties screen appears.
34. Change the name in the Project field to “S60ClientServerLab” (without the quotes) and click the Finish button to complete the import.
35. The Import dialog closes and a new project called “S60ClientServerLab” appears in the “C/C++ Projects” view.

Exercise 2 – Using the File Server

Task 2.1 – Entering Example code

1. Expand the `\S60ClientServerLab\src\S60ClientServLabContainer.cpp` file node in “C/C++ Projects” view. A list of child nodes is displayed.
2. Double click on the `S60ClientServLabContainer::WriteFileL` function (child) node. A text editor window is opened showing the contents of the file at the position of the function.
3. Add the following code to the function, as shown in bold below, to open a session with the file server:

```
void CS60ClientServLabContainer::WriteFileL()
```

```
{
RFs fs;
User::LeaveIfError(fs.Connect());
CleanupClosePushL(fs);
}
```

`RFs` is a File Server session handle. `RFs::Connect` is called to open a file server session. This function returns `KErrNone` if successful or an error code if it fails. Connection failure will cause `User::LeaveIfError` to leave. If the session is successfully opened the file server session handle is placed on the cleanup stack by the `CleanupClosePushL` function. Using this function means that our file server session handle will be automatically closed when it is popped from the cleanup stack.

4. Add some more code to the function, as shown in bold below, to create a directory:

```
...
RFs fs;
User::LeaveIfError(fs.Connect());
CleanupClosePushL(fs);

TParsePtrC parse(KDataFilePath);
TPtrC pathPtr = parse.DriveAndPath();

User::LeaveIfError(fs.MkDirAll(pathPtr));
}
```

A `TParsePtrC` object is used to extract the drive and path from `KDataFilePath`, which contains the name and path of a data file. The drive and path are extracted into a into a `TPtrC` object that is then supplied to `RFs::MkDirAll`. All the directories that do not exist in the supplied path are created by this function.

5. As shown in bold below, add further code to the function to open/create a file:

```
...
User::LeaveIfError(fs.MkDirAll(pathPtr));

RFile file;
TInt ret = file.Open(fs, KDataFilePath, EFileWrite |
EFileStreamText);

switch(ret)
      {
      case KErrNotFound:
            User::LeaveIfError(file.Create(fs, KDataFilePath,
EFileWrite | EFileStreamText));
            break;

      case KErrNone:
            // Do nothing
            break;

      default:
            User::Leave(ret);
            break;
      }

CleanupClosePushL(file);
}
```

`RFile` is a file server subsession handle to a file. The `RFile::Open` function creates the subsession and opens the data file whose name and path are contained in the `KDataFilePath` variable. The file is opened in both `EFileWrite` mode and `EFileStreamText` mode to enable us to write text to the file.

The switch statement handles the outcome of the `RFile::Open` function. If `KErrNotFound` is returned then the file does not exist and `RFile::Create` is called to create it. If creation fails then `User::LeaveIfError` will leave.

If the call to `RFile::Open` is successful (i.e returns `KErrNone`) then we just pass through the switch statement. Any other kind of failure causes `User::Leave` to be called which generates a leave exception.

Finally, the `RFile` subsession handle is pushed onto the cleanup stack.

6. Add additional code, as illustrated in bold below, to the function to write text to a file:

```
...
CleanupClosePushL(file);

TInt offset = 0;
User::LeaveIfError(file.Seek(ESeekEnd, offset));
User::LeaveIfError(file.Write(KDataEntry));

TInt fileSize;
file.Size(fileSize);

CleanupStack::PopAndDestroy(2);
}
```

`RFile::Seek` is called to set the file position (i.e. the position at which reading or writing takes place) to the end of the file. Then `RFile::Write` is called to write some text to the file. If either of these function calls fail `User::LeaveIfError` will generate a leave exception.

`RFile::Size` is called to get the size of the file in bytes, which is stored in the `fileSize` variable. Finally, the `RFile` subsession handle and the `RFs` session handle are popped from the cleanup stack and automatically closed since they are no longer required.

7. Add further code, as shown in bold below, to the function to change the label text:

```
CleanupStack::PopAndDestroy(2);

TBuf<20> numBytes;
numBytes.Format(KNumBytes, fileSize);
iTopLabel->SetTextL(KOutputFileSizeText);
iBottomLabel->SetTextL(numBytes);
}
```

The first two lines create and format a descriptor which is used in one of the calls to `CEikLabel::SetTextL`. These function-calls change the label text to display how many bytes are now in the file that was just written to.

8. Save the changes made to `S60ClientServLabContainer.cpp`.
9. Close the text editor window for `S60ClientServLabContainer.cpp`.

Task 2.2 – Building and running the application

1. Build the application for the Emulator Debug configuration of the SDK you are using by right clicking on the S60ClientServerLab project node in “C/C++ Projects” view and select the “Build Project” or “Rebuild Project” menu item. The project should build successfully

with no errors.

2. Right click on the S60ClientServerLab project node in “C/C++ Projects” view and select the “Run As > Run Symbian OS Application” menu item. The emulator will launch.

3. Follow the instructions for this step that are specific to the SDK you are using:

   S60 3rd Edition SDK and S60 3rd Edition MR SDK

   After the emulator has booted the Applications menu is shown. The S60ClientServerLab application is located in the “Installat.” folder. Navigate to this folder using the navigation keys and open it using the selection key.

   S60 3rd Edition FP1 SDK

   After the emulator has booted the Standby application is shown. Switch to the Applications menu by selecting the Applications key. The S60ClientServerLab application is located in the “Installed” folder. Navigate to this folder using the navigation keys and open it using the selection key.

4. Navigate to the S60ClientServerLab application, again using the navigation keys, and launch it using the selection key. The application is launched and displays two labels on the screen as shown in Figure 16.

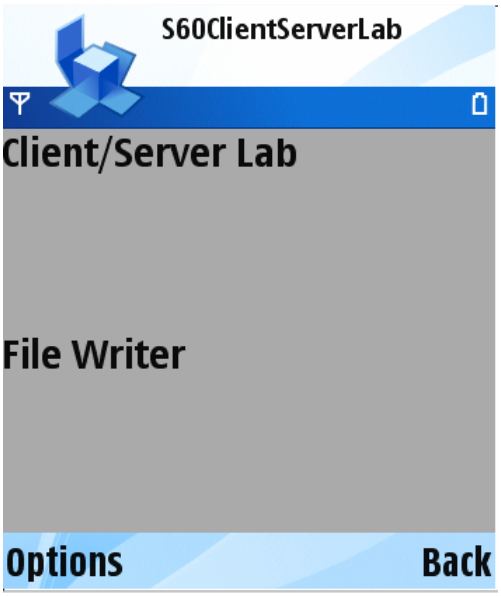


Figure 16 Screen shot of the S60ClientServerLab application just after it is launched

5. Click the Options soft-key (or press F1 on the PC keyboard). The Options menu appears as shown in Figure 17.

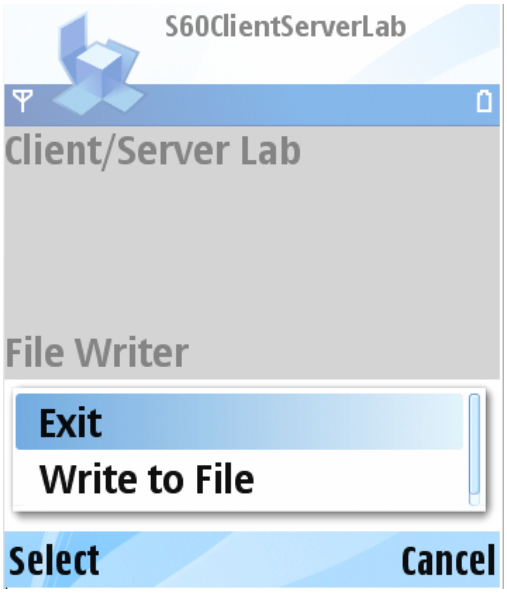


Figure 17 Screen shot of the S60ClientServerLab application just after selecting the options menu

6. Select the "Write to File" menu item (using the navigation keys and selection key). The `CS60ClientServLabContainer::WriteFileL` function is executed and the label text changes as shown in Figure 18.

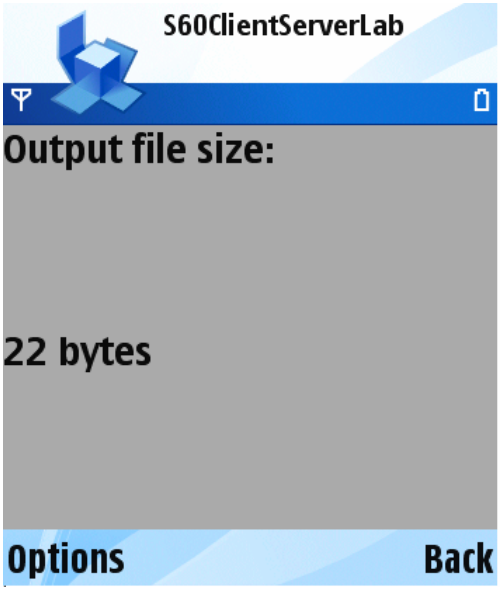


Figure 18 Screen shot of the S60ClientServerLab application just after selecting the "Write to File" menu option

7. Exit the application by selecting the Back soft-key.
8. Close the emulator.
9. Close the Carbide.c++ IDE. This completes the lab.

Developer
Training

Module #04309

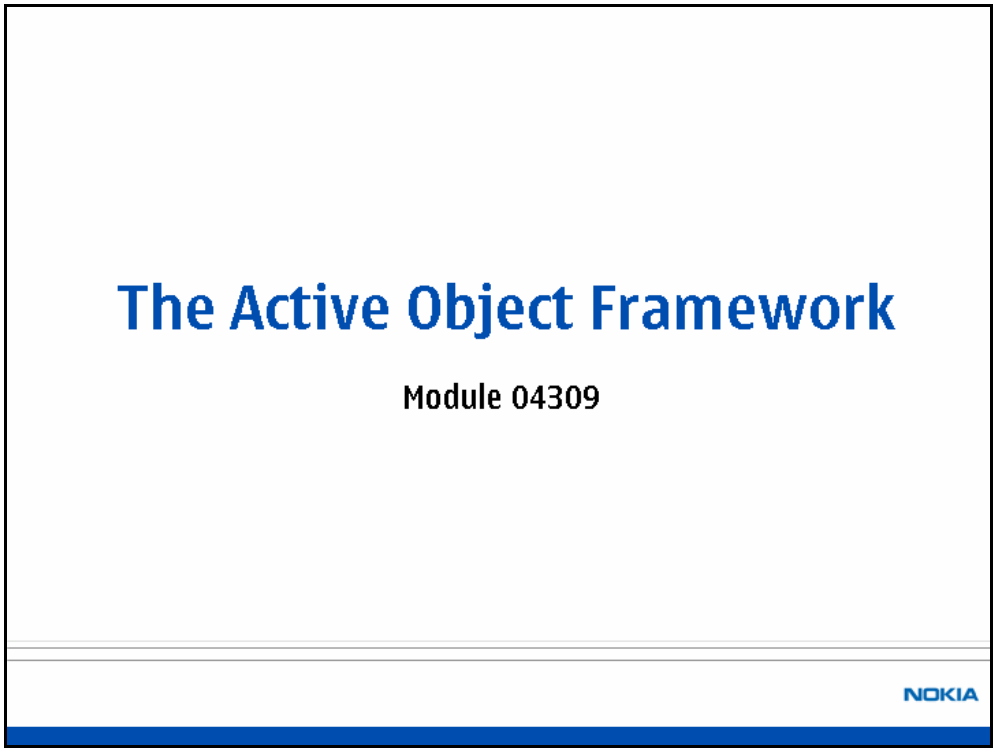
The Active Object Framework

Contents

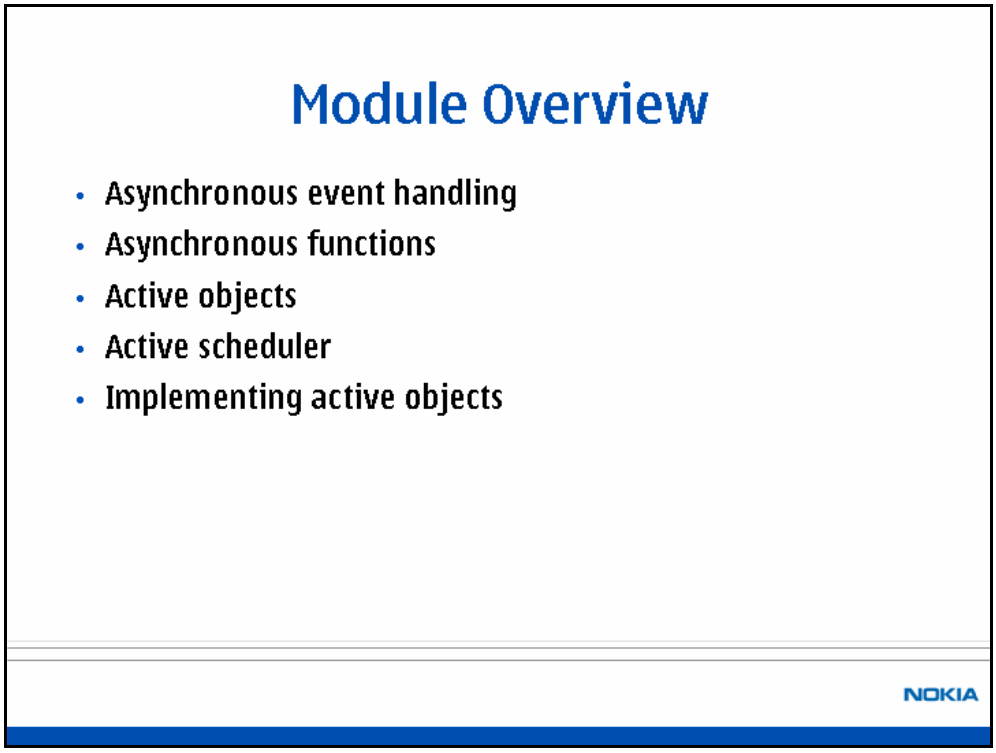
NOKIA

BO-EN-04300-20080108

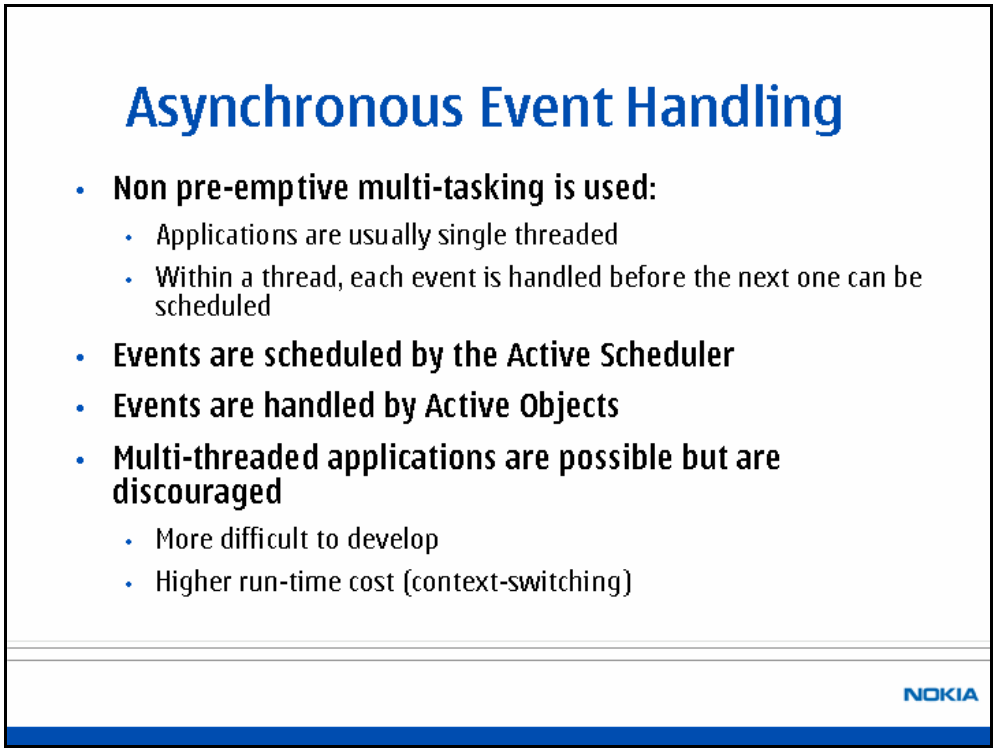
The Active Object Framework



Module Overview



Asynchronous Event Handling

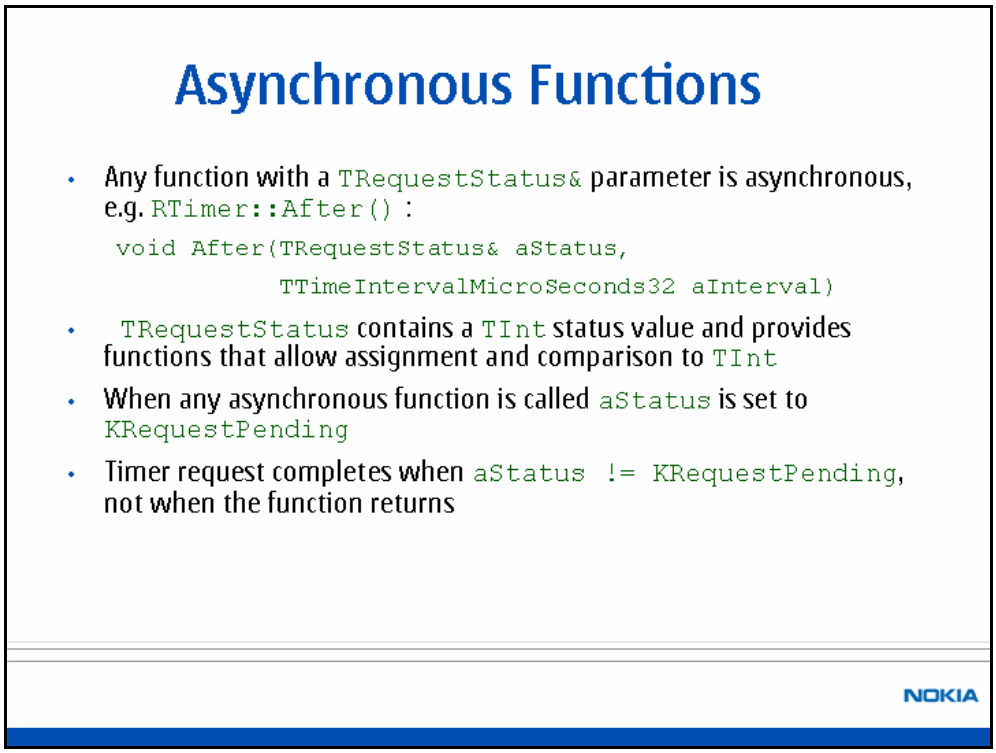


Symbian OS was designed so that applications are single threaded and spend most of the time waiting for asynchronous events to complete/occur (for example key presses). Although it is possible to use multiple threads, it is discouraged since context switching between threads incurs a higher run-time cost and will increase battery usage. Multithreaded software development is also more time consuming since issues such as accessing resources, avoiding deadlocks and inter thread communication have to be addressed.

The active object framework is used through Symbian OS to handle the asynchronous nature of the operating system. It is comprised of two classes:

- The active scheduler – This object is used to schedule events. There can only be a maximum of one per thread.
- Active objects - These objects are used to handle events. There can be many of them in any given thread.

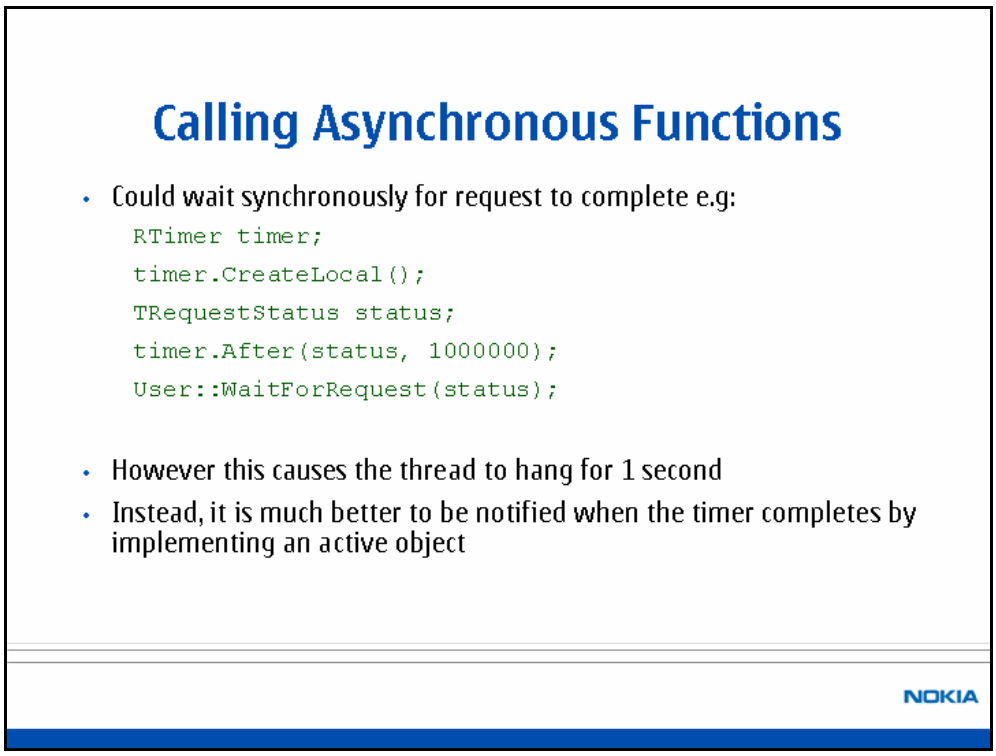
Asynchronous Functions



Symbian OS contains many APIs that are asynchronous. In the client/server module, we saw that 'R' classes access resources controlled by a server. The ones illustrated in that lesson were synchronous. However, there are also many asynchronous client/server APIs. The timer example on the slide is a rather obvious example of why a function needs to be asynchronous in a single-threaded environment. If an application was multithreaded, it would be possible to create a separate thread for the timer and make it sleep for a specified time. However in a single thread, this would prevent anything else from happening and would make the application hang from the users perspective.

In general, an API function that requires a `TRequestStatus` parameter is asynchronous (although there are exceptions to this rule; for example `User::WaitForRequest()`) and the associated operation/functionality will not complete when the function returns, but when the status value of the `TRequestStatus` parameter is set to anything other than `KRequestPending`.

Calling Asynchronous Functions

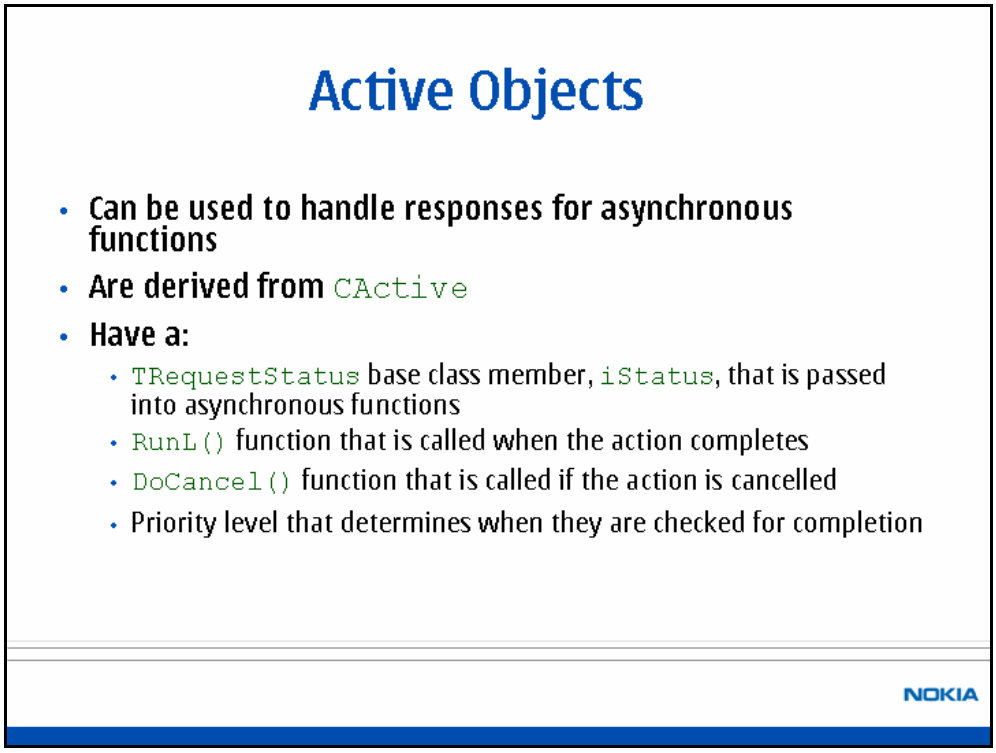


A developer may think, "Why should I be interested in the active object framework?". The example on the slide illustrates the problems faced without its use.

The example shows how to create a timer and then request a timeout value of one second. The problem with it is that it uses `User::WaitForRequest(TRequestStatus& aStatus)` to wait until the timer has completed. This means that within a single threaded application, no other events can be handled until the timer completes. For example, no key presses could be handled or redraws performed and the user would think the application had hung for a while.

A much better solution would be to receive a notification of when the timer has completed, so that in the mean time, other events could be handled. This is where active objects are used.

Active Objects



All active objects are derived from the `CActive` class at some level. The class declaration (from `e32base.h`) is included below:

```
class CActive : public CBase
{
public:
    enum TPriority
    {
    EPriorityIdle=-100,
    EPriorityLow=-20,
    EPriorityStandard=0,
    EPriorityUserInput=10,
    EPriorityHigh=20,
    };

public:
    IMPORT_C ~CActive();
    IMPORT_C void Cancel();
    IMPORT_C void Deque();
    IMPORT_C void SetPriority(TInt aPriority);
    inline TBool IsActive() const;
    inline TBool IsAdded() const;
    inline TInt Priority() const;

protected:
    IMPORT_C CActive(TInt aPriority);
    IMPORT_C void SetActive();
    virtual void DoCancel() =0;
    virtual void RunL() =0;
    IMPORT_C virtual TInt RunError(TInt aError);
```

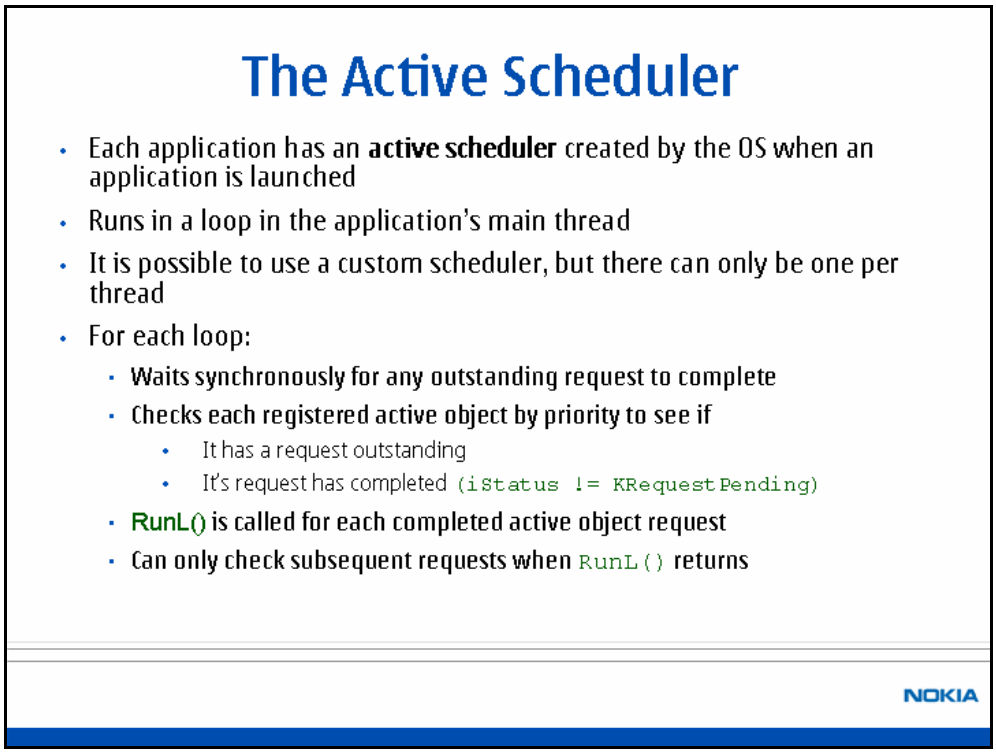
```
public:
    TRequestStatus iStatus;

private:
    TBool iActive;
    TPriQueLink iLink;
    friend class CActiveScheduler;
    friend class CServer;
};
```

Many of the class members listed above are of little concern to the developer. The most notable and commonly used ones are covered below:

The protected constructor takes a parameter to a priority level. This means that all active objects must specify the priority that they are scheduled by the active scheduler. This is usually set to `EPriorityStandard`. Unless the event to be handled is very important, the level shouldn't be set higher than this since it will alter the responsiveness of the key press handling which is undesirable.

- `RunL()` is a pure virtual function that must be implemented by the derived class. It handles the completion of the asynchronous request. This function should be small so that control is passed back to the active scheduler as soon as possible.
- `DoCancel()` is a pure virtual function that must be implemented by the derived class. Asynchronous requests made by the active object should be cancelled here.
- The `Cancel()` function calls the derived class implementation of `DoCancel()` if there is a request outstanding. It also marks the `iStatus` member as being complete (that is not equal to `KRequestPending`).
- `RunError()` can be overridden to perform cleanup in the event that the active objects `RunL()` function leaves.
- `SetActive()` needs to be called when an asynchronous function has been called. It sets the `iActive` member variable to `ETrue`. If this is not done then `RunL()` will never be called by the active scheduler.
- `IsActive()` returns the state of the `iActive` member.
- `iStatus` is passed to the `TRequestStatus&` parameter of asynchronous functions.

The Active Scheduler



The active scheduler is responsible for notifying the appropriate active object that a request has completed. It is created by the operating system when an application is launched and it runs in the application's main (and usually only) thread. It runs in a loop and waits for any request to complete, using `User::WaitForAnyRequest()`. This function blocks until a `TRequestStatus` variable in the current thread completes. When this occurs, all `TRequestStatus` variables need to be checked to see which one did complete. All registered active objects are checked in order of their priority levels to see if:

- A request is outstanding (that is `CActive::IsActive()` returns `ETrue`).
- The `iStatus` member is not set to `KRequestPending`.

If these conditions are met, the active objects `RunL()` function is called. No more requests can be processed until this function returns. The active scheduler traps any leave condition that may occur in `RunL()` and the active object's `RunError()` virtual function is called.

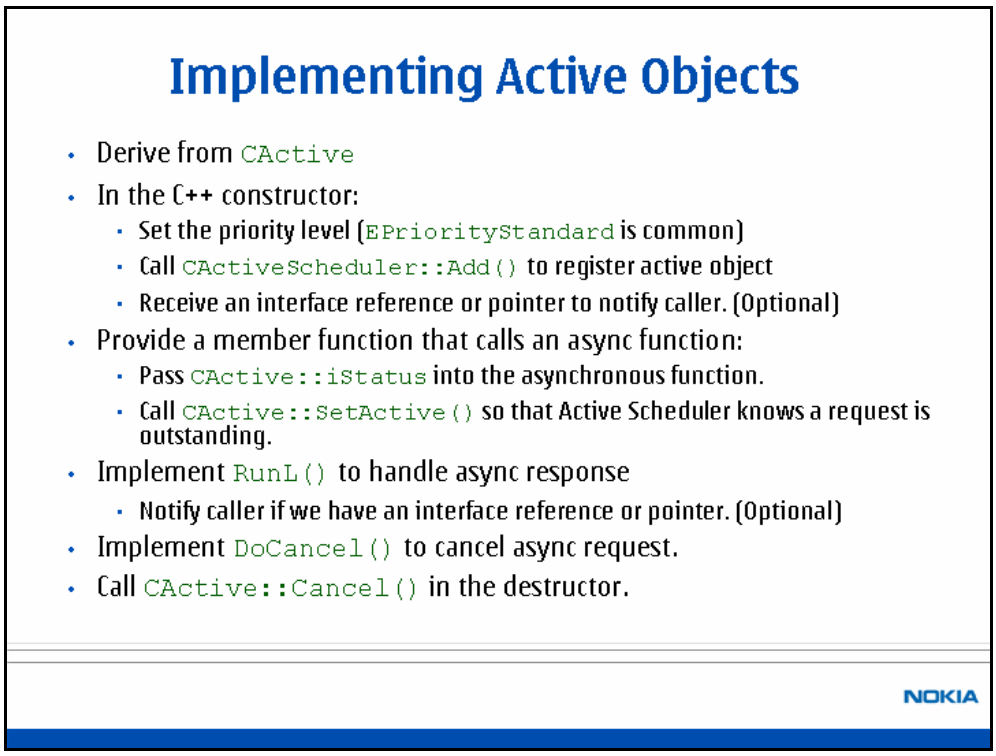
The active scheduler handles all the events for a thread, including key events. It is interesting to note that the `CCoeEnv` class, which is present in the application framework, is an active object. The active scheduler routes key press events through to this, which in turn are routed through to the App UI as menu events or to various controls/containers as key presses.

The API to the active scheduler is contained in the `CActiveScheduler` class , although since there is one already instantiated for each application, the only function likely to be called is:

```
static void Add(CActive* aActive)
```

This function is used to register an active object with the active scheduler. It is static so no instance of the active scheduler is required to call it.

Implementing Active Objects

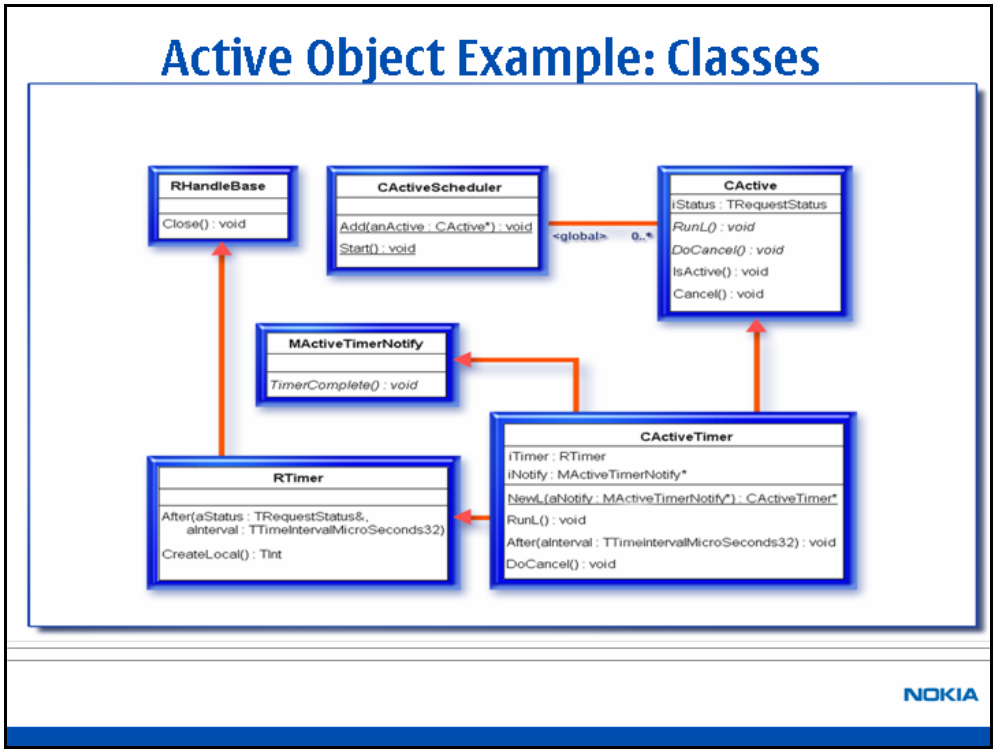


The slide illustrates the steps that must be followed in order to implement an active object. Note that:

- The `CActive` constructor is called in the initialiser list of the active object's C++ constructor, and takes the priority level parameter.
- The active scheduler will not be able to find the active object when a request completes and Panic 46 will be raised if (1) `CActiveScheduler::Add()` is not called prior to the asynchronous function call, or (2) `CActive::SetActive()` is not called after the asynchronous function call.
- An interface pointer or reference can be very useful to notify calling code that a request has completed. For instance if a class that is derived from a class other than `CActive` needs to call an asynchronous function, it is not possible for that class to become an active object itself (because inheritance from more than one C class is not allowed in Symbian OS). Therefore the client class should implement a notification interface so it can receive a callback from a contained active object. Also note that if a pointer to an interface is to be used in the active object, consider passing this into the second phase constructor so that it can leave if the pointer is NULL.
- A member function that calls the encapsulated asynchronous function is needed if the active object is to be used by another class.
- Both `RunL()` and `DoCancel()` need to be implemented in the class derived from `CActive` because they are pure virtual functions. In the `RunL()` implementation, the `iStatus` member should be checked to see if the request completed successfully. `TRequestStatus` provides a `==` operator, so if `iStatus == KErrNone`, the request has been successful. Appropriate action can be taken depending on the `iStatus` value.

- The destructor should call `CActive::Cancel()` to cancel any outstanding requests. It should also close any handles to any resources used, for example `RTimer::Close()`.

Active Object Example: Classes



The following code shows how to implement an active object that encapsulates a call to the `RTimer::After()` function. This is the preferred alternative to waiting synchronously for the timer to complete that was shown a few slides earlier. The class diagram on the slide has been simplified to show the only salient members.

First of all an interface is declared so the class that calls the active object can implement it and receive notification of the timer completing. The `TimerComplete()` function could optionally have an error code parameter.

```
class MActiveTimerNotify
    {
    public:
        virtual void TimerComplete() = 0;
    };
```

Next the active object is declared. A static two phase constructor is provided that takes a pointer to the implemented interface. Note that a reference could also be passed. The `After()` function allows the calling class to request the timeout.

```
class CActiveTimer : public CActive
    {
public: // construction / destruction
    static CActiveTimer* NewL(MActiveTimerNotify* aNotify);
    ~CActiveTimer();

public: // new functions
    After(TTimeIntervalMicroSeconds32 anInterval);

protected: // CActive overrides
    void RunL();
    void DoCancel();
```

```
protected: // construction
    CActiveTimer();
    void ConstructL(MActiveTimerNotify* aNotify);

protected: // data
    RTimer iTimer;
    MActiveTimerNotify* iNotify;
    };
```

The implementation looks like the following:

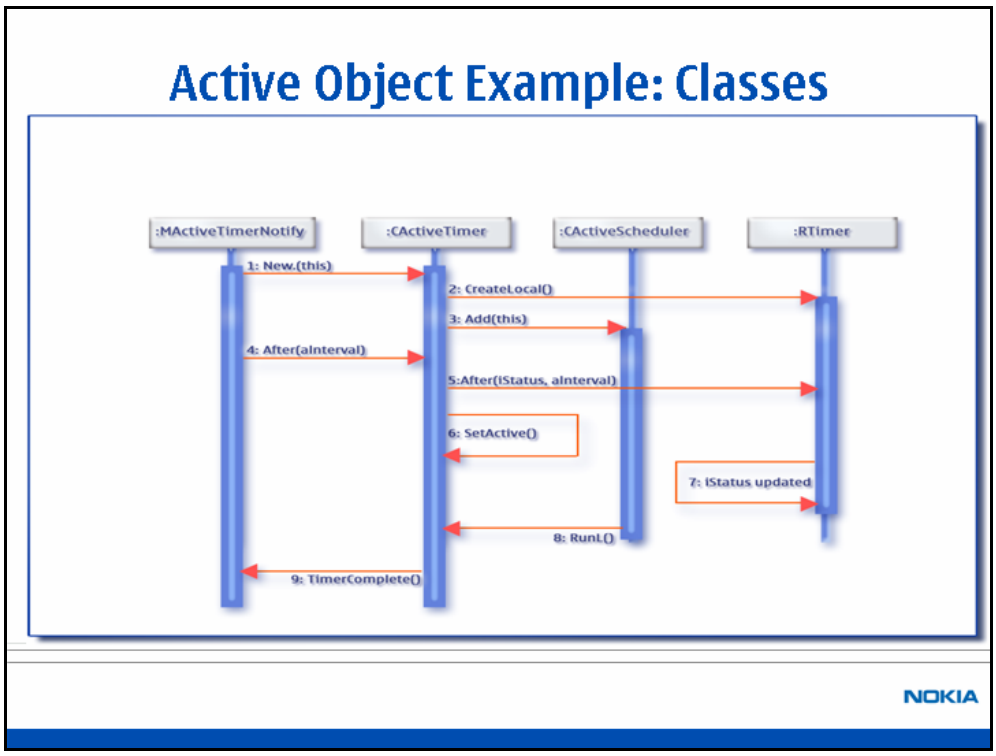
```
#include "ActiveTimer.h"

CActiveTimer* CActiveTimer::NewL(MActiveTimerNotify* aNotify)
    {
    CActiveTimer* self = new (ELeave) CActiveTimer;
    CleanupStack::PushL(self);
    self->ConstructL(aNotify);
    CleanupStack::Pop();
    return self;
    }

CActiveTimer::~CActiveTimer()
    {
    Cancel();
    iTimer.Close();
    }

void CActiveTimer::After(
    TTimeIntervalMicroSeconds32 anInterval)
    {
    iTimer.After(iStatus, anInterval);
    SetActive();
    }

void CActiveTimer::RunL()
    {
    // notify using object
    if (iStatus == KErrNone)
        {
        iNotify->TimerComplete();
        }
    else
        {
        // perform error handling e.g:
        User::Leave(iStatus.Int());
        // or could notify caller with an error code by adding
        // a parameter to MActiveTimerNotify::TimerComplete()
        }
    }

void CActiveTimer::DoCancel()
    {
    iTimer.Cancel();
    }

CActiveTimer::CActiveTimer() : CActive(EPriorityStandard)
    {
    // this active object is of standard priority
    }

void CActiveTimer::ConstructL(MActiveTimerNotify* aNotify)
    {
```

```
    User::LeaveIfNull(aNotify);
    User::LeaveIfError(iTimer.CreateLocal());
    iNotify = aNotify;
    CActiveScheduler::Add(this);
    }
```

Note that in `CActiveTimer::After()` the `TRequestStatus&` parameter used with `RTimer::After()` belongs to the base class (`CActive`).

As previously mentioned `CActiveTimer::RunL()` is called when the timer times out. This invokes a function in the using object. Note that there is already a Symbian OS class `CPeriodic` that is an active object wrapper of a timer, although it provides repeat timer functionality (rather than one off). The code above is merely a simple illustration of how to implement active objects.

Active Object Example: Classes



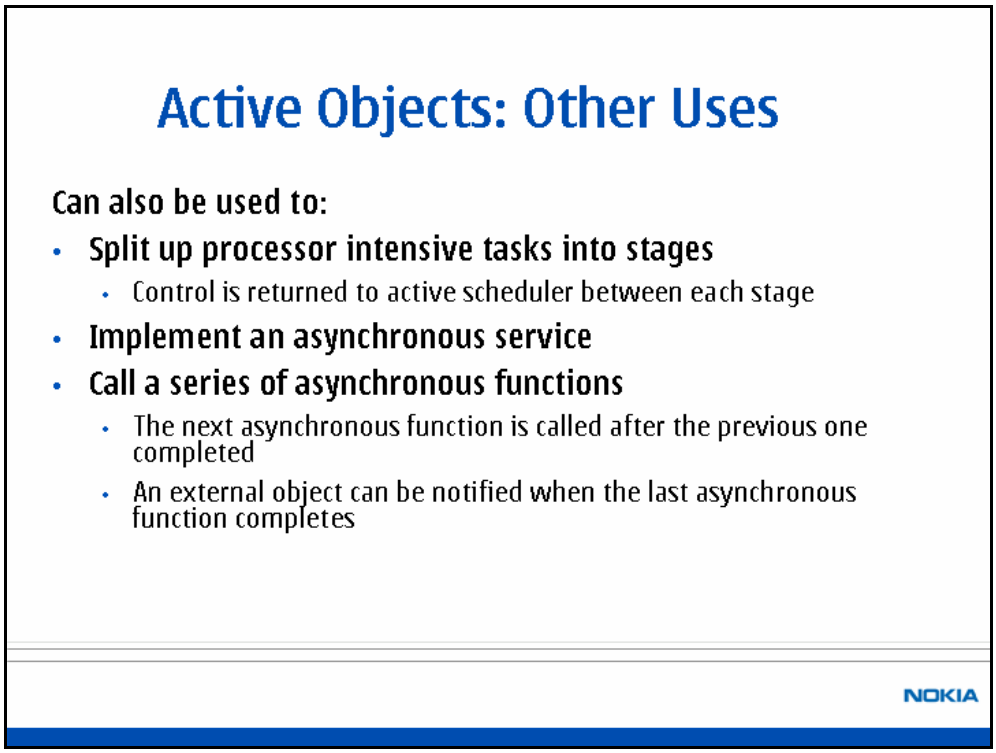
To use the timer active object, the calling class needs to derive from the `MActiveTimerNotify` interface and implement the `TimerComplete()` function. It should also have a `CActiveTimer*` member variable to store the created active object. The sequences on the slide are explained below:

1. An instance of the `CActiveTimer` class is created by calling the two-phase static constructor. This is done within the calling class implementation. Since the calling class derives from `MActiveTimerNotify`, the calling class's `this` pointer can be passed into the `CActiveTimer` constructor.
2. `CActiveTimer::ConstructL()` creates a timer via the contained `RTimer` handle.
3. `CActiveTimer::ConstructL()` registers the active object with the active scheduler.
4. The `MActiveTimerNotify` implementation initiates the timer request passing in a microsecond timeout value.
5. The `CActiveTimer::After()` implementation calls the `RTimer::After()` function passing its `iStatus` member and the timeout value. The `iStatus` value will be set to `KRequestPending` upon calling this function.
6. `CActive::SetActive()` is called so the active scheduler knows there is a request outstanding.
7. The `RTimer` instance sets the value of `CActiveTimer::iStatus` to `KErrNone` when the timeout value has been successfully reached. This is done by calling the following `User` member function: `static void RequestComplete(TRequestStatus*& aStatus, TInt aReason);`

8. The `User::WaitForAnyRequest()` call within the active scheduler's processing loop finishes. All the registered active objects are checked to see which `iStatus` value completed. `CActiveTimer::RunL()` is called.

9. `CActiveTimer::RunL()` calls the `MActiveTimerNotify` implementations `TimerComplete()` function.

Active Objects: Other Uses



The most common use of active objects is, like the example shown in the previous slides, to handle the completion of an asynchronous function call. However there are other uses which are shown on the slide. They are explained in more detail below:

If a processor-intensive task, for example numerous mathematical calculations, is allowed to run for too long, the active scheduler would not be able to process any events until the task has completed. To get around this problem, the task could be split up into stages within an active object, so that after each stage control can be returned to the active scheduler. The code would look something like the following:

```
void CActiveTask::Start()
    {
    // do Stage 1 of task
    ...
    // Return control to active scheduler
    iStageCompleted = EFirst;
    SetActive();
    User::RequestComplete(&iStatus, KErrNone);
    }

void CActiveTask::RunL()
    {
    switch (iStageCompleted)
        {
        case EFirst:
            // do Stage 2 of task
            ...
            // Return control to active scheduler
            iStageCompleted = ESecond;
            SetActive();
            User::RequestComplete(&iStatus, KErrNone);
            break;
```

```
        case ESecond:
            // do Stage 3 of task
            ...
            // Return control to active scheduler
            iStageCompleted = EThird;
            SetActive();
            User::RequestComplete(&iStatus, KErrNone);
            break;
        ...
        }
    ...
    }
```

The `Start()` function initiates the task and all other stages are then performed in the `RunL()` function. In between function calls, the active scheduler will process higher priority tasks.

Implementing an Asynchronous Service

Occasionally, you may see examples that implement active objects to provide an asynchronous service. The active object would provide a function that takes a `TRequestStatus&` parameter and then stores it in a `TRequestStatus&` member variable. When the asynchronous service completes, `User::RequestComplete()` is called on the stored reference. There are disadvantages with this approach:

- The code that calls the asynchronous function provided will need to be an active object as well to handle the response. It would be much more convenient to define a callback interface.
- The active object that implements the asynchronous service should call another asynchronous API. If it does not and it is running in the same thread as the application, no other events will be processed by the active scheduler until the asynchronous service completes.

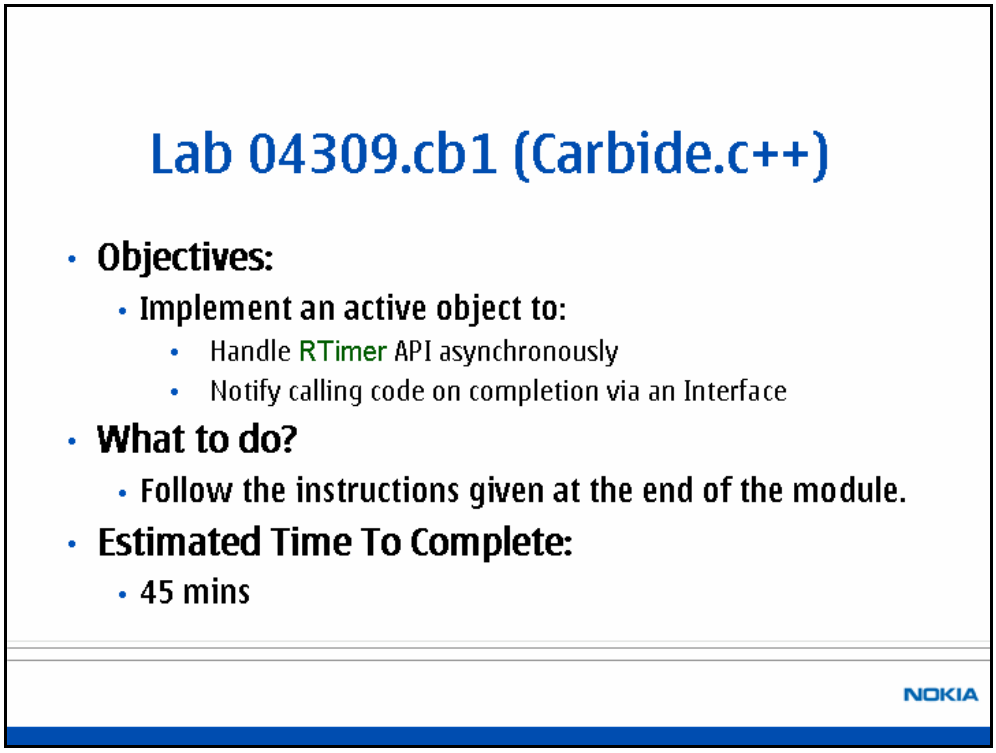
Calling More than One Asynchronous Function

This is the most common other use of active objects. It is sometimes useful to encapsulate more than one asynchronous function within an active object and notify the user when all have completed. The code for this approach is similar to the "split up tasks into stages" example in that a state machine is used. The differences are:

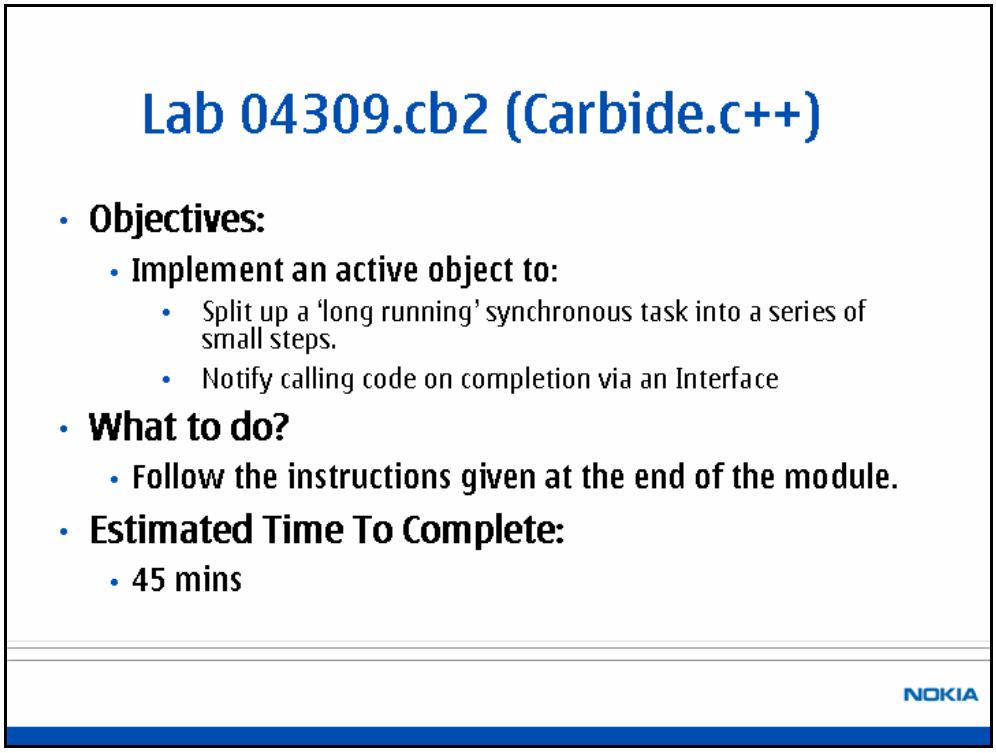
- An asynchronous function is called with the `iStatus` parameter instead of performing task stage n.
- `User::RequestComplete()` is not called after `SetActive()`.

An example of this approach would be a "communications" active object that made an external connection to another device, and then transmitted and received data. The caller would only be notified when the whole operation was completed.

Lab 04309.cb1 (Carbide.c++)



Lab 04309.cb2 (Carbide.c++)



Lab 04309.cb1

Overview

Title:	Using an Active Object to Handle an Asynchronous API Call		
Overview:	The objective of this lab is to illustrate how an active object typically handles asynchronous API calls. An asynchronous call to the `RTimer` class is used to illustrate this. The active object acts as a timer object, encapsulating calls to the `RTimer` class. The timer object is used by an S60 3rd Edition application that flashes text on and off the screen when selected by a menu item. The starting point of the Lab (after the preliminary steps) is a partially implemented application. Following the Lab instructions, a student is shown how to complete the implementation and run the application. Where appropriate, the Lab instructions also provide some background information relevant to the particular task being undertaken.		
Objectives:	To understand how to: • Use an active object to handle an asynchronous API call.		
Compatible IDE(s):	Carbide.c++ v1.2		
Compatible SDK(s):	S60 3rd Edition, S60 3rd Edition (MR), S60 3rd Edition FP1		
App. Type:	Standard GUI	**App. Name:**	AOLabTextFlash.exe
Starter Code Provided:	Yes	**Solution Code Provided:**	Yes
Estimated Time To Complete:	45 minutes		

Lab Instructions

Exercise 1 – Preliminary Steps

Task 1.1 – Checks

1. Ensure that a compatible SDK has been installed. A list of compatible SDKs to use with this lab is given in the table in the Introduction.
2. Ensure that a compatible IDE has been installed. A list of compatible IDEs to use with this lab is given in the table in the Introduction.
3. Ensure that the lab files are located in the folder `C:\Labs\Lab_04309.cb1\`. If this is not the case please refer to the setup guide for details of how to obtain them.

Task 1.2 – Make sure the emulator is not in text shell mode

1. Open Windows Explorer.
2. Navigate to the file named epoc.ini, located in the `<EPOCROOT>\epoc32\data` folder. Here, `<EPOCROOT>` is a placeholder for the root folder location of your SDK.

3. Open the epoc.ini file using Notepad.
4. Check that the file does not contain the "textshell" statement. If the statement is present in the file, then either delete the entire line containing the statement or comment the statement out. The latter is done by inserting a # symbol at the start of the line containing the statement.
5. Press 'Ctrl' + 'S' to save the contents of the file (if you altered it) and close Notepad.
6. Close Windows Explorer.

Task 1.3 – Carbide Setup

1. Launch the Carbide.c++ IDE. Depending on your Carbide.c++ settings the IDE will either display the "Workspace Launcher" dialog or open the last used workspace. If the latter occurs open the "Workspace Launcher" dialog by selecting the "File -> Switch Workspace..." menu item.
2. In the workspace launcher window, create a new workspace by typing `C:\Labs\<Workspace>` in the Workspace field. Here `<Workspace>` is a placeholder for a folder name of you choosing that does not already exist and does not contain spaces.
3. Click the OK button.
4. Close the Welcome window after it appears when the workspace is opened.

Task 1.4 – Importing the project

36. In the Carbide.c++ IDE, select the "File > Import" menu item. The Import dialog appears.
37. Select "Symbian OS bld.inf file" from the "Symbian OS" subfolder as the import source and click the Next button. A new screen appears on the Import dialog.
38. Enter `C:\Labs\Lab_04309.cb1\starter\group\bld.inf` in the "bld.inf file" field. Alternatively, click the "Browse..." button and then navigate to and select the correct file via the dialog. When this is done a list of configurations appears in the "SDKs and Build Configurations pane".
39. Expand the list of configurations for your installed SDK and tick the checkbox next to the "Emulator Debug" configuration. (Note that other configurations may be chosen in addition to this one, but they are not used in this lab.) Then select the "Next" button.
40. Ensure the MMP file and all the make files are selected on the "MMP Selection" screen. Then select the Next button. The Project Properties screen appears.
41. Change the name in the Project field to "AOLabTextFlash" (without the quotes) and click the Finish button to complete the import.
42. The Import dialog closes and a new project called "AOLabTextFlash" appears in the "C/C++ Projects" view.

Exercise 2 – Completing the implementation

Task 2.1 – Completing the class definition of CActiveTimer

1. Double click on the `\AOLabTextFlash\inc\ActiveTimer.h` file node in C/C++ Projects view. A text editor window is opened showing the contents of the file.
2. Add a forward declaration of the `MActiveTimerNotify` class, as illustrated by the code in bold below. This allows the class to be referenced in this header file without including its class definition.

```
// Copyright (c) 2006 Nokia Corporation.
```

```
#ifndef _ACTIVE_TIMER_H
#define _ACTIVE_TIMER_H

#include <e32base.h>

class MActiveTimerNotify;

class CActiveTimer : public CActive
...
```

3. Add a `MActiveTimerNotify&` parameter to the two-phase constructor as shown by the code in bold below. This will allow the `CActiveTimer` class to notify the class that constructed it.

```
...
class CActiveTimer : public CActive
    {
    public: // construction / destruction

        static CActiveTimer* NewL(MActiveTimerNotify& aNotifier);
        ~CActiveTimer();
...
```

4. Add a `MActiveTimerNotify&` parameter to the C++ constructor as shown by the code in bold below. This will allow the `CActiveTimer` class to notify the class that constructed it.

```
...
    protected: // construction

        CActiveTimer(MActiveTimerNotify& aNotifier);
        void ConstructL();
...
```

5. Add two new member variables; one of type `RTimer` and one a `MActiveTimerNotify` reference. This is shown by the code in bold below.

```
...
    protected: // data

        RTimer iTimer;
        MActiveTimerNotify& iNotifier;
    };
...
```

6. Select 'Ctrl' + 'S' to save the contents of the editor window and close it.

Task 2.2 – Implementing CActiveTimer construction

1. Double click on the `\AOLabTextFlash\src\ActiveTimer.cpp` file node in C/C++ Projects view. A text editor window is opened showing the contents of the file.
2. Add a `MActiveTimerNotify&` parameter to the `CActiveTimer::NewL` function and pass the `MActiveTimerNotify&` parameter to the C++ constructor. This is shown by the code in bold below.

```
ActiveTimer* CActiveTimer::NewL(MActiveTimerNotify& aNotifier)
    {
    CActiveTimer* self = new (ELeave) CActiveTimer(aNotifier);
    CleanupStack::PushL(self);
```

```
...
```

3. Perform the following additions to the C++ constructor. First add a `MActiveTimerNotify&` parameter to the function. Next, add code to initialise the `iNotifier` member variable by adding it to the initialiser list. Then add code to register the `CActiveTimer` object with the active scheduler. All these changes are shown in the code in bold below.

```
CActiveTimer::CActiveTimer(MActiveTimerNotify& aNotifier) :
    CActive(EPriorityStandard),
    iNotifier(aNotifier)
    {
    CActiveScheduler::Add(this);
    }
```

4. In the `CActiveTimer::ConstructL` function construct the `iTimer` member by calling `RTimer::CreateLocal` and leave if there is an error. This is illustrated by the code in bold below.

```
void CActiveTimer::ConstructL()
    {
    User::LeaveIfError(iTimer.CreateLocal());
    }
```

Task 2.3 – Implementing timer request and timer completion

1. In the `CActiveTimer::After` function call `RTimer::After` on the `iTimer` member variable, passing `iStatus` and `anInterval` as parameters. Also call `CActive::SetActive` to notify the application's active scheduler that an asynchronous request is outstanding. The following code in bold illustrates this.

```
void CActiveTimer::After(TTimeIntervalMicroSeconds32 anInterval)
    {
    iTimer.After(iStatus, anInterval);
    SetActive();
    }
```

2. In the `CActiveTimer::RunL` function, call `MActiveTimerNotify::TimerComplete` on the `iNotifier` member passing `iStatus.Int()` as a parameter. This notifies the code that constructed this class when the timer has completed. The following code in bold illustrates this.

```
void CActiveTimer::RunL()
    {
    iNotifier.TimerComplete(iStatus.Int());
    }
```

3. In the `CActiveTimer::DoCancel` function, call `RTimer::Cancel` on the `iTimer` member to cancel the outstanding timer request. The following code in bold illustrates this.

```
void CActiveTimer::DoCancel()
    {
    iTimer.Cancel();
    }
```

Task 2.4 – Implements the active object destructor

1. In the `CActiveTimer` destructor, call `CActive::Cancel` to cancel any outstanding request. Remember that `CActiveTimer` is derived from `CActive`. Also, call `Close` on the `iTimer` member to free resources. The following code in bold illustrates both these

changes.

```
CActiveTimer::~CActiveTimer()
    {
    Cancel();
    iTimer.Close();
    }
```

2. Select 'Ctrl' + 'S' to save the contents of the editor window and close it.

Task 2.5 – Using the active object

1. Double click on the `\AOLabTextFlash\src\AOLabTextFlashContainer.cpp` file node in "C/C++ Projects" view. A text editor window is opened showing the contents of the file.
2. In the `CAOLabTextFlashContainer::ConstructL` function add code to create the active object as illustrated by the code in bold below.

```
void CAOLabTextFlashContainer::ConstructL(const TRect& aRect)
    {
    CreateWindowL();

    iTopLabel = new (ELeave) CEikLabel;
    iTopLabel->SetContainerWindowL(*this);
    iTopLabel->SetTextL(KHelloText);

    iBottomLabel = new (ELeave) CEikLabel;
    iBottomLabel->SetContainerWindowL(*this);
    iBottomLabel->SetTextL(KWorldText);

    iActiveTimer = CActiveTimer::NewL(*this);
    iIsVisible = ETrue;

    SetRect(aRect);
    ActivateL();
    }
```

3. In the `CAOLabTextFlashContainer` destructor, add code to delete the active object as shown by the code in bold below.

```
CAOLabTextFlashContainer::~CAOLabTextFlashContainer()
    {
    delete iActiveTimer;
    delete iTopLabel;
    delete iBottomLabel;
    }
```

4. In the `CAOLabTextFlashContainer::FlashingText` function, add code to handle the call to start the active timer object as shown by the code in bold below. Note that the `CAOLabTextFlashContainer::FlashingText` function is called from `CAOLabTextFlashAppUi::HandleCommandL` to begin flashing the text on screen and `CAOLabTextFlashAppUi::HandleCommandL` is itself called by the application framework in response to the user selecting a menu item. In the code `KTimeoutValue` is the timeout value for the active object timer; i.e. the frequency of the flashing.

```
void CAOLabTextFlashContainer::FlashingText()
    {
    iTopLabel->SetTextL(KHelloText);
    iBottomLabel->SetTextL(KWorldText);
    DrawNow();

    if (!iIsFlashing)
        {
        iIsVisible = ETrue;
        iIsFlashing = ETrue;
        iActiveTimer->After(KTimeoutValue);
        }
    }
```

5. In the `CAOLabTextFlashContainer::TimerComplete` function, add code to restart the active timer object. This is shown by the code in bold below. Note that this function is a callback function that is called by the active timer object when it completes.

```
void CAOLabTextFlashContainer::TimerComplete(TInt aError)
    {
    if (KErrNone == aError)
        {
        iIsVisible = !iIsVisible;
        iActiveTimer->After(KTimeoutValue);
        }
    else
...
```

6. In the `CAOLabTextFlashContainer::StopFlashing` function, add code to cancel the active timer object. This is shown by the code in bold below.

```
void CAOLabTextFlashContainer::StopFlashing()
    {
    if (iIsFlashing)
        {
        iIsFlashing = EFalse;
        iActiveTimer->Cancel();
        }

    iTopLabel->MakeVisible(ETrue);
    iBottomLabel->MakeVisible(ETrue);
    }
```

7. Select 'Ctrl' + 'S' to save the contents of the editor window and close it.

Task 2.6 – Building and running the application

1. Build the application for the Emulator Debug configuration of the SDK you are using by right clicking on the AOLabTextFlash project node in "C/C++ Projects" view and select the "Build Project" or "Rebuild Project" menu item. The project should build successfully with no errors.
2. Right click on the AOLabTextFlash project node in "C/C++ Projects" view and select the "Run As > Run Symbian OS Application" menu item. The emulator will launch.
3. Follow the instructions for this step that are specific to the SDK you are using:

   S60 3rd Edition SDK and S60 3rd Edition MR SDK

After the emulator has booted the Applications menu is shown. The A0LabTextFlash application is located in the “Installat.” folder. Navigate to this folder using the navigation keys and open it using the selection key.

S60 3rd Edition FP1 SDK

After the emulator has booted the Standby application is shown. Switch to the Applications menu by selecting the Applications key. The A0LabTextFlash application is located in the “Installed” folder. Navigate to this folder using the navigation keys and open it using the selection key.

Figure 19 Screen shot of the A0LabTextFlash application just after it is launched.

4. Navigate to the A0LabTextFlash application, again using the navigation keys, and launch it using the selection key. The application is launched and displays two labels on the screen as shown in Figure 19.

Figure 20 Screen shot of the A0LabTextFlash application just after selecting the options menu

5. Click the Options soft-key (or press F1 on the PC keyboard). The Options menu appears as shown in Figure 20.
6. Select the “Start Flashing” menu item. The text on the screen begins flashing on and off. The screen when the text disappears is shown in Figure 21. When the text reappears the screen looks like it did when the application was launched (see Figure 19).

Figure 21 Screen shot of the AOLabTextFlashapplication when the text has flashed off

7. Click the Options soft-key (or press F1 on the PC keyboard). The Options menu appears again as shown in Figure 5, but this time has the option to “Stop Flashing” instead of “Start Flashing”.

Figure 22 Screen shot of the AOLabTextFlash application just after selecting the options menu (when the text is flashing).

8. Select the “Stop Flashing” menu item. The text on the screen stops flashing and the

application looks like it did when it was launched (see Figure 19).

9. Exit the application by selecting the Exit soft-key.
10. Close the emulator.
11. Close the Carbide.c++ IDE. This completes the lab.

Lab 04309.cb2

Overview

Title:	Using an Active Object to Split up a Long-Running Synchronous Task into a Series of Small Steps		
Overview:	The objective of this lab is to illustrate how an active object can be used to split up a 'long running' synchronous task into a series of small steps. The 'long running' synchronous task we have used to illustrate this is a simple bubble sorting algorithm. Numbers are read from an input file, which are then sorted using the algorithm and then written back to an output file. The active sorting object is used by an S60 3rd Edition application that provides a menu item to begin (or cancel) sorting and displays text to inform the user of the sorting status. The starting point of the Lab (after the preliminary steps) is a partially implemented application. Following the Lab instructions, a student is shown how to complete the implementation and run the application. Where appropriate, the Lab instructions also provide some background information relevant to the particular task being undertaken.		
Objectives:	To understand how to: • Use an active object to split a long running task into a series of steps.		
Compatible IDE(s):	Carbide.c++ v1.2		
Compatible SDK(s):	S60 3rd Edition, S60 3rd Edition (MR), S60 3rd Edition FP1		
App. Type:	Standard GUI	**App. Name:**	AOLabBubbleSort.exe
Starter Code Provided:	Yes	**Solution Code Provided:**	Yes
Estimated Time To Complete:	45 minutes		

Lab Instructions

Exercise 1 – Preliminary Steps

Task 1.1 – Checks

1. Ensure that a compatible SDK has been installed. A list of compatible SDKs to use with this lab is given in the table in the Introduction.
2. Ensure that a compatible IDE has been installed. A list of compatible IDEs to use with this lab is given in the table in the Introduction.
3. Ensure that the lab files are located in the folder `C:\Labs\Lab_04309.cb2\`. If this is not the case please refer to the setup guide for details of how to obtain them.

Task 1.2 – Make sure the emulator is not in text shell mode

1. Open Windows Explorer.
2. Navigate to the file named epoc.ini, located in the `<EPOCROOT>\epoc32\data` folder. Here, `<EPOCROOT>` is a placeholder for the root folder location of your SDK.
3. Open the epoc.ini file using Notepad.
4. Check that the file does not contain the "textshell" statement. If the statement is present in the file, then either delete the entire line containing the statement or comment the statement out. The latter is done by inserting a # symbol at the start of the line containing the statement.
4. Press 'Ctrl' + 'S' to save the contents of the file (if you altered it) and close Notepad.
5. Close Windows Explorer.

Task 1.3 – Carbide Setup

1. Launch the Carbide.c++ IDE. Depending on your Carbide.c++ settings the IDE will either display the "Workspace Launcher" dialog or open the last used workspace. If the latter occurs open the "Workspace Launcher" dialog by selecting the "File -> Switch Workspace..." menu item.
2. In the workspace launcher window, create a new workspace by typing `C:\Labs\<Workspace>` in the Workspace field. Here `<Workspace>` is a placeholder for a folder name of you choosing that does not already exist and does not contain spaces.
3. Click the OK button.
4. Close the Welcome window after it appears when the workspace is opened.

Task 1.4 – Importing the project

43. In the Carbide.c++ IDE, select the "File > Import" menu item. The Import dialog appears.
44. Select "Symbian OS bld.inf file" from the "Symbian OS" subfolder as the import source and click the Next button. A new screen appears on the Import dialog.
45. Enter `C:\Labs\Lab_04309.cb2\starter\group\bld.inf` in the "bld.inf file" field. Alternatively, click the "Browse..." button and then navigate to and select the correct file via the dialog. When this is done a list of configurations appears in the "SDKs and Build Configurations pane".
46. Expand the list of configurations for your installed SDK and tick the checkbox next to the "Emulator Debug" configuration. (Note that other configurations may be chosen in addition to this one, but they are not used in this lab.) Then select the "Next" button.
47. Ensure the MMP file and all the make files are selected on the "MMP Selection" screen. Then select the Next button. The Project Properties screen appears.
48. Change the name in the Project field to "AOLabBubbleSort" (without the quotes) and click the Finish button to complete the import.
49. The Import dialog closes and a new project called "AOLabBubbleSort" appears in the "C/C++ Projects" view.

Task 1.6 – Copy an input data file into the application's private directory

1. Open Windows Explorer.
2. Navigate to the file named `sortdata_in.dat`, located in the `C:\Labs\Lab_04309.cb2\starter\data` folder.

3. Copy this file to the application's private directory; i.e. copy it to folder `<EPOCROOT>\epoc32\winscw\c\private\0515DFCE`, where `<EPOCROOT>` is as described above.
4. Close Windows Explorer.

Exercise 2 – Completing the implementation

Task 2.1 – Completing CActiveBubbleSorter's class definition

1. Double click on the `\AOLabBubbleSort\inc\ActiveBubbleSorter.h` file node in "C/C++ Projects" view. A text editor window is opened showing the contents of the file.
2. Add a forward declaration of the `MBubbleSortNotify` class, as illustrated by the code in bold below. This allows the class to be referenced in this header file without including its class definition.

```
// Copyright (c) 2006 Nokia Corporation.

#ifndef __ACTIVEBUBBLESORTER_H__
#define __ACTIVEBUBBLESORTER_H__

#include <e32base.h>
#include <f32file.h>

class MBubbleSortNotify;

class CActiveBubbleSorter : public CActive
...
```

3. Add a `MBubbleSortNotify&` parameter to the two-phase constructor as shown by the code in bold below. This will allow the `CActiveBubbleSorter` class to notify the class that constructed it.

```
...
class CActiveBubbleSorter : public CActive
    {
    public: // Construction/destruction
        static CActiveBubbleSorter* NewL(MBubbleSortNotify&
aNotifier);
        ~CActiveBubbleSorter();
...
```

4. Add a `MBubbleSortNotify&` parameter to the C++ constructor as shown by the code in bold below. This will allow the `CActiveBubbleSorter` class to notify the class that constructed it.

```
...
    private: // Construction

        CActiveBubbleSorter(MBubbleSortNotify& aNotifier);
        void ConstructL();
...
```

5. Add four new member variables as shown by the code in bold below: `iX` and `iY` are of type `TInt` and are used in the bubble sort algorithm, `iNumbersArray` is an integer array which contains the numbers to be sorted and `iNotifier` allows a `CActiveBubbleSorter` object to notify the code that uses it, when sorting has completed.

```
...
    private: // data

        TInt                iX;
        TInt                iY;
        RArray<TInt>        iNumbersArray;
        MBubbleSortNotify&  iNotifier;
    };

#endif // __ACTIVEBUBBLESORTER_H__

// End of file
```

6. Select 'Ctrl' + 'S' to save the contents of the editor window and close it.

Task 2.2 – Implementing CActiveBubbleSorter construction

1. Double click on the `\AOLabBubbleSort\src\ActiveBubbleSorter.cpp` file node in "C/C++ Projects" view. A text editor window is opened showing the contents of the file.
2. Add a `MBubbleSortNotify&` parameter to the `CActiveBubbleSorter::NewL` function and pass the `MBubbleSortNotify&` parameter to the C++ constructor. This is shown by the code in bold below.

```
CActiveBubbleSorter* CActiveBubbleSorter::NewL(MBubbleSortNotify&
aNotifier)
{
    CActiveBubbleSorter* self = new (ELeave)
CActiveBubbleSorter(aNotifier);
    CleanupStack::PushL(self);
...
```

3. Perform the following additions to the C++ constructor. First add a `MBubbleSortNotify&` parameter to the function. Next, add code to initialise the `iNotifier` member variable by adding it to the initialiser list. Then add code to register the `CActiveBubbleSorter` object with the application's active scheduler. All these changes are shown in the code in bold below.

```
CActiveBubbleSorter::CActiveBubbleSorter(MBubbleSortNotify&
aNotifier) :
    CActive(EPriorityStandard),
    iNotifier(aNotifier)
    {
    CActiveScheduler::Add(this);
    }
```

Task 2.3 – Implementing timer request and timer completion

1. In the `CActiveBubbleSorter::StartL` function, add the code shown in bold below.

```
void CActiveBubbleSorter::StartL()
    {
    if (IsActive())
        {
        User::Leave(KErrInUse);
        }

    iNumbersArray.Reset();
```

```
    ReadNumbersFromFileL();

    iX = 0;
    iY = 0;

    TRequestStatus* status = &iStatus;
    User::RequestComplete(status, KErrNone);
    SetActive();
    }
```

The first part checks to make sure the active object is not already active. If the check is successful and the object is not active some initialisation takes place. First, `RArray::Reset` is called which empties the array (if necessary) so it can be reused. Then `CActiveBubbleSorter::ReadNumbersFromFileL` which reads numbers from an input file into the array. Also, the variables `iX` and `iY`, used in the sorting algorithm, are both initialised to zero.

The last part of the function causes the `RunL` function of `CActiveBubbleSorter` to be called at the earliest opportunity through the active scheduler. The calling of the `User::RequestComplete` function is like calling an instantaneous asynchronous request that does nothing. The active object is set active via the call to `CActive::SetActive`.

2. In the `CActiveBubbleSorter::RunL` function, implement the bubble sorting algorithm on a per step basis. The following code in bold illustrates this.

```
void CActiveBubbleSorter::RunL()
    {
    if (iNumbersArray[iY] > iNumbersArray[iY+1])
        {
        TInt temp = iNumbersArray[iY+1];
        iNumbersArray[iY+1] = iNumbersArray[iY];
        iNumbersArray[iY] = temp;
        }

    iY++;
    TInt count = iNumbersArray.Count();

    if (iY >= count-1)
        {
        iY = 0;

        if (iX < count)
            {
            iX++;
            }
        else
            {
            // Finished sorting
            WriteNumbersToFileL();
            iNumbersArray.Reset();
            iNotifier.SortComplete(iStatus.Int());
            return;
            }
        }
```

```
    TRequestStatus* status = &iStatus;
    User::RequestComplete(status, KErrNone);
    SetActive();
    }
```

To aid understanding, the following code shows what the bubble sort algorithm looks like in its usual form; implemented in a single step. Compare this with the above code.

```
TInt count = iNumbersArray.Count();
for (iX = 0; iX < count; iX++)
    {
    for (iY = 0; iY < count-1; iY++)
        {
        if (iNumbersArray[iY] > iNumbersArray[iY+1])
            {
            TInt temp = iNumbersArray[iY+1];
            iNumbersArray[iY+1] = iNumbersArray[iY];
            iNumbersArray[iY] = temp;
            }
        }
    }
```

The last part of the `CActiveBubbleSorter::RunL` function is called in the event another step is required to complete the sort. It causes `CActiveBubbleSorter::RunL` to be called again at the earliest opportunity through the Active Scheduler to perform the next step.

On the final step the `CActiveBubbleSorter::RunL` function calls `CActiveBubbleSorter::WriteNumbersToFileL` which writes numbers from the `iNumbersArray` to an output file. In addition the `CActiveBubbleSorter::RunL` function also notifies the code that uses the active object that the sort has completed. This is done via the call to `MActiveTimerNotify::SortComplete`.

3. In the `CActiveBubbleSorter::DoCancel` function, we must add code to tidy up the `iNumbersArray` for reuse by calling `RArray::Reset` on it and notify the code that uses the active object that the sort has been cancelled. The following code in bold illustrates this.

```
void CActiveBubbleSorter::DoCancel()
    {
    iNumbersArray.Reset();
    iNotifier.SortComplete(KErrCancel);
    }
```

Task 2.4 – Implements the active object destructor

1. In the `CActiveBubbleSorter` destructor, call `CActive::Cancel` to cancel any outstanding request. Remember that `CActiveTimer` is derived from `CActive`. Also, call Close on the `iNumbersArray` member to free resources. The following code in bold illustrates both these changes.

```
CActiveBubbleSorter::~CActiveBubbleSorter()
    {
    Cancel();
    iNumbersArray.Close();
    }
```

2. Select 'Ctrl' + 'S' to save the contents of the editor window and close it.

Task 2.5 – Using the active object

1. Double click on the `\AOLabBubbleSort\src\AOLabBubbleSortContainer.cpp` file node in "C/C++ Projects" view. A text editor window is opened showing the contents of the file.
2. In the `CAOLabBubbleSortContainer::ConstructL` function add code to create the active object as illustrated by the code in bold below.

```
void CAOLabBubbleSortContainer::ConstructL(const TRect& aRect)
    {
    CreateWindowL();

    iTopLabel = new (ELeave) CEikLabel;
    iTopLabel->SetContainerWindowL( *this );
    iTopLabel->SetTextL(KStartActionText);
    iBottomLabel = new (ELeave) CEikLabel;
    iBottomLabel->SetContainerWindowL( *this );
    iBottomLabel->SetTextL(KNullDesC);

    iActiveBubbleSorter = CActiveBubbleSorter::NewL(*this);

    SetRect(aRect);
    ActivateL();
    }
```

3. In the `CAOLabBubbleSortContainer` destructor, add code to delete the active object as shown by the code in bold below.

```
CAOLabBubbleSortContainer::~CAOLabBubbleSortContainer()
    {
    delete iActiveBubbleSorter;
    delete iBottomLabel;
    delete iTopLabel;
    }
```

4. In the `CAOLabBubbleSortContainer::SortL` function, add code to start the sorting task as shown by the code in bold below.

```
void CAOLabBubbleSortContainer::SortL()
    {
    iIsSorting = ETrue;
    iTopLabel->SetTextL(KCancelActionText);
    iBottomLabel->SetTextL(KSortingText);
    DrawNow();

    iActiveBubbleSorter->StartL();
    }
```

5. In the `CAOLabBubbleSortContainer::CancelSortL` function, add code to cancel the active object. This is shown by the code in bold below.

```
void CAOLabBubbleSortContainer::CancelSortL()
    {
    iActiveBubbleSorter->Cancel();
    iIsSorting = EFalse;

    iTopLabel->SetTextL(KStartActionText);
```

```
iBottomLabel->SetTextL(KSortingCancelled);
DrawNow();
}
```

6. Select 'Ctrl' + 'S' to save the contents of the editor window and close it.

Task 2.6 – Building and running the application

10. Build the application for the Emulator Debug configuration of the SDK you are using by right clicking on the AOLabBubbleSort project node in "C/C++ Projects" view and select the "Build Project" or "Rebuild Project" menu item. The project should compile successfully with no errors.
11. Right click on the AOLabBubbleSort project node in "C/C++ Projects" view and select the "Run As > Run Symbian OS Application" menu item. The emulator will launch.
12. Follow the instructions for this step that are specific to the SDK you are using:

    S60 3rd Edition SDK and S60 3rd Edition MR SDK

    After the emulator has booted the Applications menu is shown. The AOLabBubbleSort application is located in the "Installat." folder. Navigate to this folder using the navigation keys and open it using the selection key.

    S60 3rd Edition FP1 SDK

    After the emulator has booted the Standby application is shown. Switch to the Applications menu by selecting the Applications key. The AOLabBubbleSort application is located in the "Installed" folder. Navigate to this folder using the navigation keys and open it using the selection key.

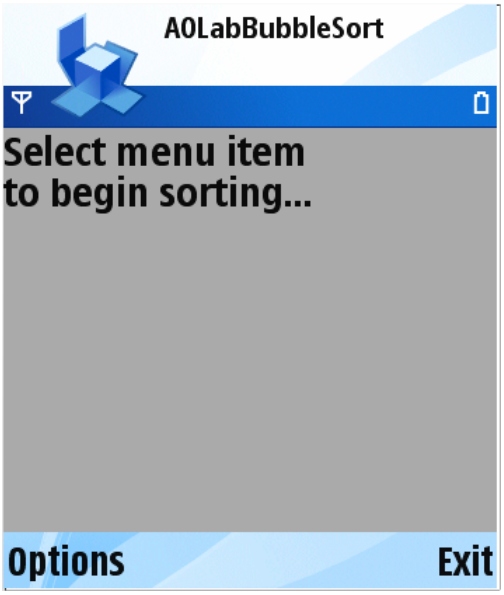


Figure 23 Screen shot of the AOLabBubbleSort application just after it is launched.

13. Navigate to the AOLabBubbleSort application, again using the navigation keys, and launch it using the selection key. The application is launched and displays the screen shown in Figure 19.

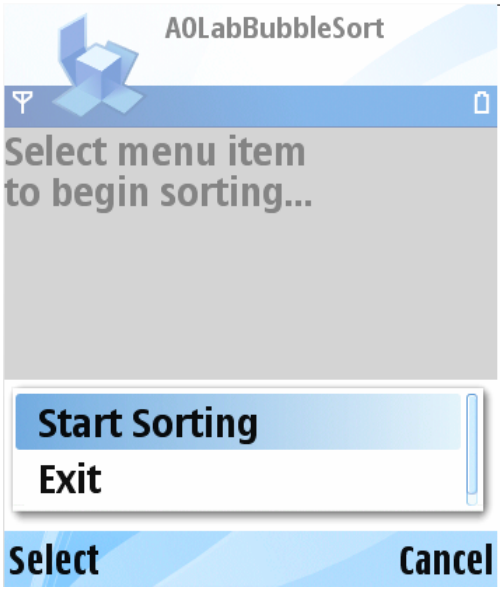


Figure 24 Screen shot of the AOLabBubbleSort application just after selecting the options menu

14. Click the Options soft-key (or press F1 on the PC keyboard). The Options menu appears as shown in Figure 20.

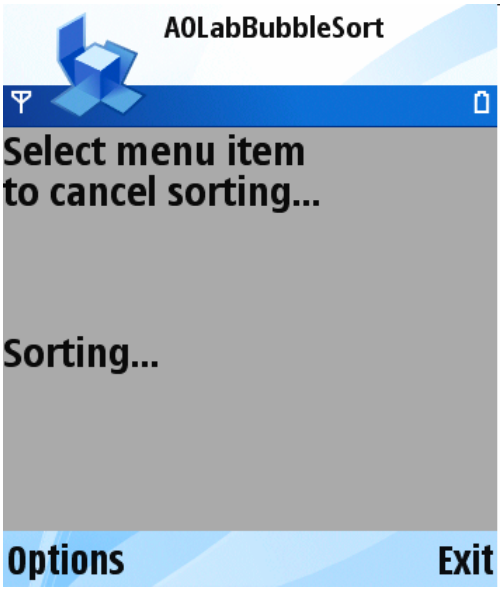


Figure 25 Screen shot of the AOLabBubbleSort application during sorting

15. Select the “Start Sorting” menu item. The text on the screen changes to indicate the status of the sort as shown in Figure 21.

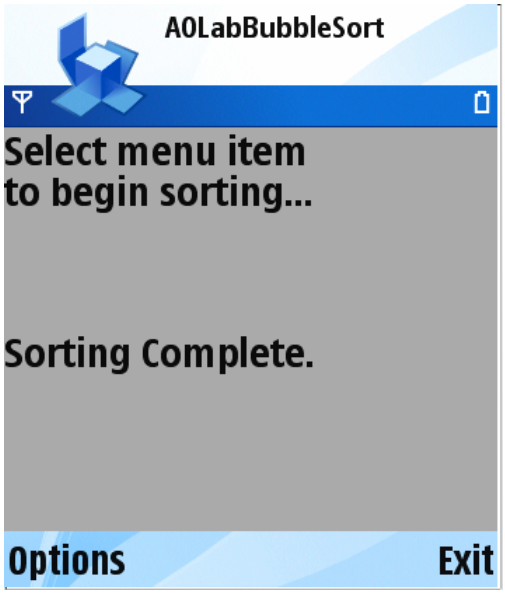


Figure 26 Screen shot of the AOLabBubbleSort application just after completing the sort successfully

16. Wait for the sort to complete. The text on the screen changes to indicate the status of the sort. If the sort completes successfully the screen in Figure 5 is shown.

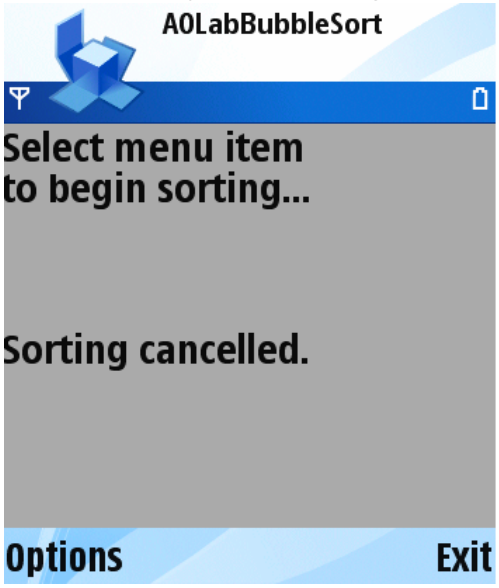


Figure 27 Screen shot of the AOLabBubbleSort application just after cancelling the sort

17. Select the Options menu and select the “Start Sorting” menu item. Before the sort completes select the Options menu again and select the Cancel Sorting menu item. The sort will be cancelled and the text on the screen changes to indicate this as shown in Figure 27.
18. Exit the application by selecting the Exit soft-key.
19. Close the emulator.
20. Exit the Carbide.c++ IDE. This completes the lab.