KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Politechnika Wrocławska

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

ROZWÓJ POTENCJAŁU I OFERTY DYDAKTYCZNEJ POLITECHNIKI WROCŁAWSKIEJ

Wrocław University of Technology

Internet Engineering

Marek Piasecki

APPLICATION PROGRAMMING - MOBILE COMPUTING

Mobile Computing

Wrocław 2011

Projekt współfinansowany ze środków Unii Europejskiej w ramach
Europejskiego Funduszu Społecznego

Wrocław University of Technology

Internet Engineering

Marek Piasecki

APPLICATION PROGRAMMING - MOBILE COMPUTING

Mobile Computing

Wrocław 2011

Reviewer: Czesław Smutnicki

ISBN 978-83-62098-25-5

Published by PRINTPAP Łódź, www.printpap.pl

Table of contents

1. The Course Book Outline

The purpose of this course handbook, is to gather together teaching and training materials from different sources (programming documentation, Web pages of manufacturers and IDE providers, technical journals, textbooks, etc.), to support students activities during "Mobile Computing" laboratory. This laboratory is planned as a supplementary class to INEA00112 academic course "Application programming – Mobile Computing" at the Wrocław University of Technology.

The course aims to give students most of the essential knowledge, skills and hands-on experience indispensable in programming advanced mobile phones, palmtops and smartphones. Apart from 30h lecture, the course comprises 15h hands-on programming laboratory and 15h of individual project activities. As a basic pre-requisite, it is expected that the course participants have a basic knowledge in the area of object-oriented programming using Java and C++ languages.

The laboratory is composed of six programming exercises which are arranged in three series presenting three totally different mobile platforms: Sun's Java Microedition (MIDlets), Microsoft Windows Mobile and Nokia's Symbian S60 with Qt/QtMobility framework. The principle idea of such selection, is to create effective learning process with slowly stepping-up curve of acquired knowledge difficulty (from more familiar to more demanding). The second idea is to present mobile technologies which are the most popular in the area of European Union. This is a reason why some less popular platforms like BlackBerry or Palm were omitted.

The course starts with the Java ME framework, which should be the easiest to learn, because students are expected to be familiar with Java language and Java technologies from the 1st semester course "Application programming - Java and XML technologies". Additionally, almost 100% of mobile phones support Java ME MIDlets, so all the students will have the occasion to test the created software on their own mobile devices, which can make the laboratory more attractive.

The second series of exercises is related to mobile incarnation of Microsoft .NET framework – Windows Mobile. The C# programming language will be used in Windows Mobile section, but all examples are constructed in such a way that they should be easy readable and understandable for people familiar with Java. Besides, Microsoft Windows products are very popular in Europe and many students are familiar with .NET technologies and creating Windows.Forms based UI for desktop applications. Normally, Microsoft development environment for Windows Mobile is an expensive commercial product. It can be even more expensive if we try to create real business applications utilizing enterprise solutions like Web Services, Microsoft SQL Server or Microsoft Internet Information Server. Fortunately, the Faculty of Electronics is a member of MSDN AA program, and the course participants can use these tools without charge (which could be perceived as an additional benefit).

The last series of exercises is devoted to Nokia's Symbian, which has been the most popular mobile platform in the world, and especially in Europe. Cumulative number of mobile devices shipped with the Symbian OS, is estimated at about 390 million, which generates an impressive market of software-consumers. Unfortunately, native Symbian C++ dialect with its a little antiquated software techniques developed two decades ago (for the much more restricted mobile hardware) caused tiresome complexity of the created software code. In two beginning exercises, we will shortly present some of these techniques, to give the student "a taste" of traditional Symbian programming in Carbide.C++ development environment. Final exercises will present very attractive current Nokia's approach to Symbian and Maemo/MeeGo platform programming, based on Qt and QtMobility frameworks.

All the three chosen platforms have a settled reputation, confirmed by more than 10 years of evolutionary development, multitude of shipped devices, with complete and widely accessible documentation.

We hope, that such selection and composition of teaching materials, based around a series of practical exercises, will create a good starting point for further individual studies, which are indispensable in the emerging world of mobile devices.

In the last three years, smartphones have transformed from a niche product category to a fast growing segment, playing a key role in competitive struggle between mobile and Internet giants. The unprecedented success of iPhone, followed shortly after by Google Android, changed market requirements almost overnight. Today smartphones are all about smooth delivery of digital content, applications and Web services.

Most of recently emerged mobile platforms, especially iPhone and Android, are subject to instant changes. It is considerably probable that in two years some of their today's documentation will become out-of-date. To avoid this problem, we will suggest them as the subject of an individual student's project, which will be the third supplementary class form of this Mobile Computing course.

About the Author

Marek Piasecki is an academic lecturer at the Institute of Computer Engineering, Control and Robotics, Wrocław University of Technology. A graduate in: Automation (MS), Computer Science (MS) and Mobile Robotics (PhD). The area of his scientific interests includes: software engineering for mobile devices, autonomous and adaptive systems, user modeling and user adapted interaction, personalization, automatic recommendation, computer ontologies, semantic Web, software agent systems and languages, infobots, soft computing and artificial intelligence.

2. Programming Java Micro Edition (J2ME)

Many mobile applications use the Java 2.0 Micro Edition (Java ME, J2ME) platform, which was initially developed by Sun for small devices like mobile phones, but is now used on a wide variety of devices. Examples of such devices are: TV set-top boxes, Internet TVs, car computers, phones, pagers, Personal Data Assistants. Java ME uses scaled down subsets of Java Standard Edition (J2SE) components, virtual machines and APIs. It also defines several APIs that are specifically targeted at consumer mobile and embedded devices. The most popular variant of J2ME platform is a Mobile Information Device Profile (MIDP). An application written using the MIDP APIs is called MIDlet, and is directly portable to any MIDP device according to general Java motto "Write Once, Run Anywhere".

Important aspect of J2ME is its support from huge Java community and its security. MIDlet suites can be cryptographically signed and verified on the device, which gives users some security about executing downloaded code. A permissions architecture controls an application access to critical API, allowing the user to deny untrusted code access to certain device resources. For example, enables to block network connections if it is not explicitly necessary. J2ME is deployed globally on millions of phones and PDAs, and is supported by most of leading integrated development environments.

2.1 Understanding J2ME, Configurations and Profiles

J2ME is divided into configurations, profiles, and optional APIs, which provide specific information about APIs and different categories of devices. A configuration defines a Java Virtual Machine for a specific family of devices, based on memory constraints and processor performance. It specifies a subset of the full Java 2 Platform Standard Edition (J2SE) APIs, that will be used on the mobile device. Currently there are two: the Connected Device Configuration (CDC) and the Connected, Limited Device Configuration (CLDC). Device manufacturers are responsible for porting a specific configuration to their devices. In the course of our laboratory we will concentrate on devices with CLDC configuration.

CLDC configuration encompasses mobile phones, pagers, PDAs, and other small devices having: limited display and input, limited memory (160KB to 512KB), limited CPU power, limited network throughput (data rates starting from 9.6Kbps.) and limited battery life. CLDC 1.1 includes some enhancements to CLDC 1.0, including support for floating-point data types.

Profiles are more specific than configurations. They provide additional APIs, such as user interface, necessary to develop applications running on the device. A profile is a high level abstraction of all resources for a class of devices which can be used by an application. It includes APIs for application life cycle, user interface, and persistent storage. Several different profiles are being developed under the Java Community Process. For example, the PDA Profile (PDAP) is designed for advanced palmtop devices with an onboard memory 512KB÷16MB and an application model using a subset of the J2SE Abstract Windowing Toolkit (AWT) for graphic user interface. But at the moment, only one of them, the Mobile Information Device Profile (MIDP) is the most frequently met on the market.

There are two versions of MID profile: basic MIDP 1.0 (JSR 37), and extended MIDP 2.0 (JSR 118) which features a number of enhancements (e.g. support for multimedia, game API, HTTPS connection). During this course, we will concentrate on MIDP 2.0, which has the following requirements:

- A minimum of 256KB of ROM for the MIDP implementation
- A minimum of 128KB of RAM for the Java runtime heap
- A minimum of 8KB of nonvolatile writable memory for persistent data
- A screen of at least 96×54 pixels
- User input by keypad, keyboard, or touch screen
- Two-way network connection

More information about MIDP could be found at http://java.sun.com/products/midp/. The APIs available to a MIDP application come from packages in both CLDC and MIDP (Figure 2-1).

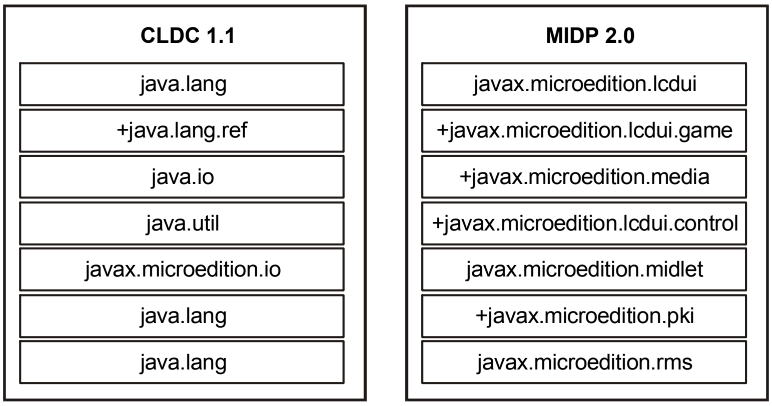


Figure 2-1 Java packages composing basic J2ME/MIDP 2.0 application programming interface

Optional packages provide functionality that may not be included in a specific configuration or profile. One example of an optional package is the Bluetooth API [BTAPI] described in JSR-82, providing a standardized API for Bluetooth networking, which could be used on devices equipped with Bluetooth transmitter. Another example is JSR-75 [PIM & File Data API] which enables personal information management (phone contacts, calendar events, alarms, etc) and direct access to the device file system and additional memory cards.

2.2 Tools – J2ME Programming Environments

Although the MIDlets are designed to run on a small pocket-size devices, they are programmed on regular desktop computers. There is a very wide choice of different IDEs supporting MIDlet programming under a variety of OSes. Because J2ME creation was led by Sun, the first programming tool could be Sun's J2ME Wireless Toolkit, available from http://java.sun.com/products/j2mewtoolkit/. Unfortunately, this WTK toolkit does not contain a specialized programming editor or other important developing tools enabling advanced debugging of MIDlet code. Among other possible IDEs, the most popular are (in alphabetical order):

- Borland JBuilder X Mobile Edition,
- Eclipse J2ME Plugin,
- IBM WebSphere Studio Device Developer,
- NetBeans Mobility,
- Nokia Developer's Suite for J2ME,
- Research In Motion BlackBerry Java Development Environment,
- Sun Java Studio Mobility.

In the course of this laboratory, NetBeans will be suggested as a professional and very comfortable J2ME programming environment (See figure 2-2). As an alternative to NetBeans, the Eclipse could be chosen. In general, NetBeans is an IDE for developing Java server and client applications. By additional installation of NetBeans Mobility Pack plug-in, its functionality is extended to Java Microedition development (NetBeans version 6.0 and above comes integrated with mobility pack). NetBeans with Mobility Pack supports two J2ME configurations and two additional embedded platforms:

- Connected Limited Device Configuration (CLDC). Which includes support for the area of our interest: Mobile Information Device Profile (MIDP).
- Connected Device Configuration (CDC). Supporting advanced smart phones, set-top boxes, embedded servers and devices.
- JavaFX Mobile. Supporting mobile, desktop, web and television screens.
- Java Card platform which enables application development for smart cards and other microdevices.

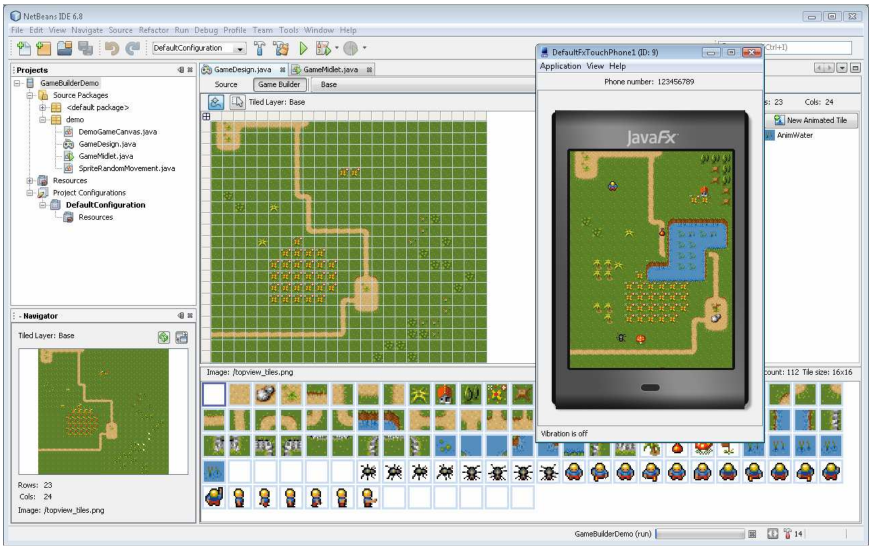
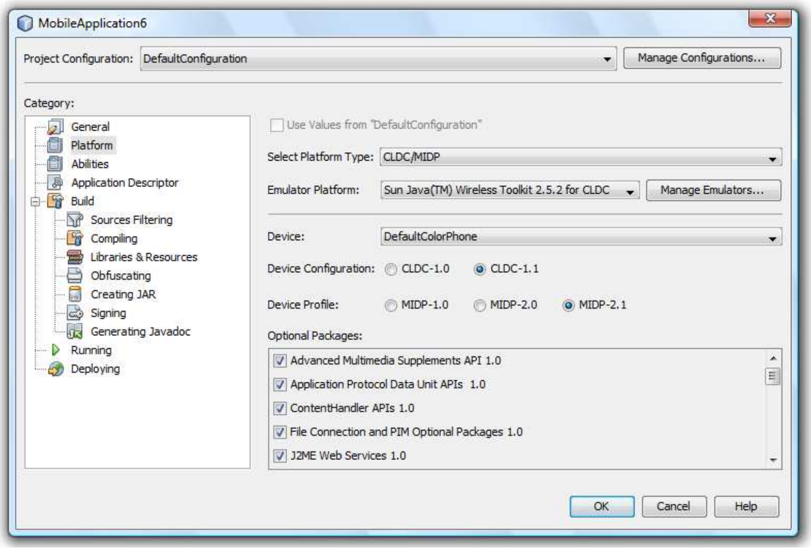


Figure 2-2 Game Builder - example screen of NetBeans IDE programmer interface

The first step in developing MIDlet applications is to create a new Project, which contains information about application components: program files, resources and IDE configuration settings. To create a project, select File→New Project from the main menu of NetBeans IDE. Then select Java ME in the project Categories window and choose Mobile Application (for totally new code) or alternatively Mobile Project with Existing MIDP Sources (for importing from existing code). In the case of a new project, following dialog windows of project wizard enables to set the project name, location, select emulator platform, and give the names of new MIDlet class and package name. Project properties can be browsed and modified by selecting File→Project Properties (Figure 2-3). Using this option, we can review and edit: included optional packages, application descriptor attributes for JAD and JAR manifest, additional libraries, JAR names, MIDlet signing settings, and many other.


MobileApplication6

Project Configuration: DefaultConfiguration

Category:
- General
- Platform
- Abilities
- Application Descriptor
- Build
  - Sources Filtering
  - Compiling
  - Libraries & Resources
  - Obfuscating
  - Creating JAR
  - Signing
  - Generating Javadoc
- Running
- Deploying

[ ] Use Values from "DefaultConfiguration"

Select Platform Type: CLDC/MIDP

Emulator Platform: Sun Java(TM) Wireless Toolkit 2.5.2 for CLDC

Device: DefaultColorPhone

Device Configuration: CLDC-1.0 [✓ CLDC-1.1]

Device Profile: MIDP-1.0 MIDP-2.0 [✓ MIDP-2.1]

Optional Packages:
- [✓] Advanced Multimedia Supplements API 1.0
- [✓] Application Protocol Data Unit APIs 1.0
- [✓] ContentHandler APIs 1.0
- [✓] File Connection and PIM Optional Packages 1.0
- [✓] J2ME Web Services 1.0

Figure 2-3 NetBeans IDE MIDP project properties dialog.

After finishing the MIDlet code editing, the project should be compiled by selecting Run→Build Project option. To run the MIDlet on the mobile device emulator, select Run→Run Main Project option from main IDE menu (or Debugg Main Project to initialize debugging). You will see a mobile phone emulator window with the screen showing a list of MIDlets in the downloaded MIDlet suite (in most cases there will be only one). Finally, click on the LAUNCH soft button to start up the MIDlet, and see the first MIDlet screen. To finalize testing, select the MIDlet's EXIT command to leave the MIDlet, and then close the emulator window or hit the ESC key, to end the emulator session

One or more MIDlets can be packaged in a single jar file and distributed as a MIDlet suite. All of the MIDlets in a suite are installed onto a device as a single entity and can share theirs resources. A MIDlet suite consists of Java Application Descriptor file (.jad) and Java Archive file (.jar). The JAD file includes the archive file name, the names and classes for each MIDlet in the suite, and more. The JAR file contains the MIDlet classes and resource files (data, icons, sounds, etc.). In NetBeans IDE, the

JAD and JAR files for the MIDlet suite will be generated and placed in the "dist" subdirectory of the project.

The final step of our application development is a MIDlet suite deployment, i.e. the process of transferring the application developed in the IDE to a mobile device. NetBeans IDE supports several methods of deployment, as an example:

- Device Anywhere Deployment - website for professional mobile developers that allows you to remotely deploy MIDlet suites to the most popular mobile devices on the market today. (Registration required).
- File Transfer Protocol (FTP) - uses the file transfer protocol to move the MIDlet suite files to a remote server over the Internet.
- Nokia Mobile Devices - transfers the MIDlet suite files to a Nokia Mobile device from the IDE using Nokia's PC Suite.
- Sony Ericsson Phone - transfers the MIDlet suite files to a Sony Ericsson phone. This deployment method also enables you to run and debug the on-device application from the IDE.
- Windows Mobile Devices - transfers the MIDlet suite files to a Windows Mobile device using ActiveSync and Sun's CLDC/MIDP Virtual Machine.

We may also use any file transfer tool (via USB cable or Bluetooth connection) to copy the JAR/JAD files to the physical mobile device memory.

2.3 Programming Exercises

In the following sections, we are going to illustrate the MIDlet development process with five practical programming exercises. These exercises are planned to be performed in the course of the three beginning laboratory classes. Every exercise is supported with brief theoretical introduction and contains sample code listings. In first exercises, the laboratory participants will get familiar with MIDlet life cycle, creation of user friendly interface from high-level components and generation of two-dimensional graphic. Fourth exercise concerns examples of Wireless Messaging Services based on SMS interception. The last exercise is devoted to MIDlet's capabilities of persistent data storage.

Exercise J2ME.ex1 Standard MIDlet Life Cycle

Create a simple application (with a very limited or without user interface), demonstrating all possible states and transpositions of MIDlet's life cycle:

a) Pure background MIDlet (console) application, which does not possess any graphical user interface components, which notifies life cycle changes by `System.out.print()` messages in debug/output window of IDE.

b) "HelloWorld" application displaying a single `Form` with the selected title, which quits after counting down assigned time interval (e.g. 5 seconds).

MIDP applications are called MIDlets, as continuation of the Java naming pattern from applets and servlets. To be classified as a `MIDlet`, a mobile Java application must extend the abstract class `javax.microedition.midlet.MIDlet`, which controls the MIDlet lifecycle. Moreover it has to be packaged and distributed in a Java Archive (JAR) file which includes a MANIFEST.MF file.

The core of the MIDlet lifecycle is the Application Management Software (AMS),which is a part of the device's operating environment and manages MIDlets states. As the result of MIDlet launching, the AMS instantiates it by calling the method, which starts its lifecycle. Then AMS maintains control over the MIDlet lifecycle throughout its execution. As a response to user initiated actions, or other events (like a phone call) the MIDlet can be send to the background (paused state). The system can close a MIDlet at any time. Just before closing, the AMS calls the MIDlet's `destroyApp()` method, then waits about 5 seconds to give it a chance to save resources, and finally terminates the MIDlet forcefully.

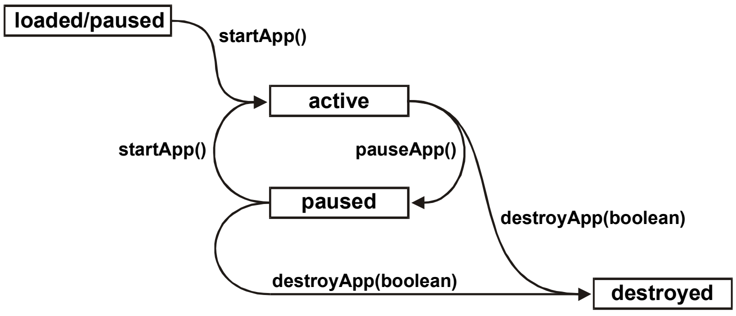


Figure 2-4 The life cycle of the MIDlet [5]

To organize a MIDlet life cycle, it was decided that MIDlet can exist in four different states: loaded, active, paused and destroyed. Figure 2-4 gives an overview of the MIDlet lifecycle and Listing 2-1 illustrates the skeleton of typical MIDlet class code implementation.

Listing 2-1 Skeleton of typical MIDlet class implementation

```
import javax.microedition.midlet.*;

public class ExampleMIDlet extends MIDlet {
    public ExampleMIDlet() {
        System.out.println(">>> Entering Constructor");
        // . . .
    }

    public void startApp() {
        System.out.println(">>> Entering StartApp");
        // . . .
    }

    public void pauseApp() {
        System.out.println(">>> Entering PauseApp");
        // . . .
    }

    public void destroyApp(boolean unconditional) {
        System.out.println(">>> Entering destroyApp");
        // . . .
    }
}
```

When a MIDlet is loaded onto the device, its constructor is called, and the MIDlet enters the loaded state. When a user launches the MIDlet, the program manager (AMS) starts the application by calling the `startApp()` method. After `startApp()`, the MIDlet enters the active state, and hold it until the program manager calls `pauseApp()` or `destroyApp()`. All state change callback methods should perform as fast as possible, because the state is not changed before the method returns. In the `pauseApp()` method, the MIDlet should release resources that are not needed while paused, to avoid resource conflicts with other applications and to reduce battery power consumption. Calling `destroyApp()` method, indicates that the MIDlet process should terminate. The MIDlet can request that it does not enter the destroyed state by throwing a `MIDletStateChangeException`. This only is a valid response if the unconditional flag is set to false.

State changes could be also initialized by the MIDlet itself. Being in active state, MIDlet can call `notifyPaused()` method, which notify the application manager that it decides to go to the paused state. Being in paused state, the MIDlet can request to resume its activity by calling `resumeRequest()` (which would result with `startApp()` callback, if AMS decides to activate this application). In order to force termination, a MIDlet can call `notifyDestroyed()` method, which notifies the application management software that it has entered into the destroyed state. Be careful, in this case, the application management software will not call the `destroyApp()` callback method, and the MIDlet must have performed all the cleaning operations by itself. Termination by calling `System.exit()`, known from standard edition, is not supported in MIDP. The MIDlet, can also send itself to the background by calling `Display.setCurrent(null)` method (described in the second exercise). In fact, this command will not change the current displayable, but will be interpreted by AMS as a request from the application to be placed into the background. In similar manner, to activate from the background, the MIDlet can call `Display.setCurrent(Displayable)`, where `Displayable` is a reference to window object, which should be shown on the device screen.

Listing 2-2 Example implementation of ”Hello World” flash-up MIDlet

```
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;

public class HelloWorld extends MIDlet {

   public void startApp() {
       Display.getDisplay(this).setCurrent(new Form("Hello World"));  ①

        // 1 second delay
        try{ Thread.sleep(1000); }                                    ②
        catch(InterruptedException e){}

        notifyDestroyed();                                            ③
    }

    public void pauseApp() {}

    public void destroyApp(boolean unconditional) {}
}
```

Listing 2-2 presents more interesting example code of flash-up "Hello World" MIDlet, which: ① creates user interface window entitled "Hello World", than waits some time interval (in milliseconds) ②, and finally calls `notifyDestroyed()` method ③, to inform AMS about its lifecycle termination. In a more correct MIDlet implementation, operations ② and ③ should be transferred to a separate thread (because `startApp()` method should not contain any time consuming elements). This separate thread could randomly call all possible MIDlet's signaling methods: `notifyPaused(), resumeRequest(), notifyDestroyed().` The resulting messages, notifying about state changes, could be observed in output console window of NetBeans IDE.

Exercise J2ME.ex2
MIDlet User Interface and Input Event Handling

Create a MIDlet application utilizing MIDP High-Level UI API to compose user friendly interface for standard key-pad based cell phone.

a) MIDlet implementation of currency exchange calculator.
b) MIDlet implementation of questionnaire form, collecting selected personal, education and employment data for national census
(for future sending, by SMS or WLAN, to census office)
c) The user interface for a foreign language mini-dictionary for a tourist, supporting self-memorizing and quick searching for popular phrases.

The user interface requirements for mobile devices are different from those for desktop computers. The display size of handheld devices is smaller, and the input does not always include pointing devices like a mouse or a stylus. We cannot follow the same user interface programming guidelines for applications running on desktop computers and hand-held devices. The MIDP creators decided not to translate the existing Java standard edition UI packages, like AWT or Swing, for J2ME. The reason is, that mentioned packages are optimized for desktop computers. They support many features not found on handheld devices. For example, extensive support for window management, such as resizing overlapping windows. However, the limited display size of handheld devices makes resizing a window impractical. Additionally, the shape of mobile application window has to reflect the shape of the device screen, which is usually diverse for different device classes. Simultaneously, the application interface has to adopt to dynamical changes of mobile device screen orientation (vertical or horizontal).

To achieve high portability between different devices, the API employs a high-level abstraction and gives you little control over its look and feel. For example, you cannot define the visual appearance (shape, color, or font) of components. It is the device manufacturer responsibility to define the exact implementation of components appearance and manage all necessary adaptation to the device's hardware and native user interface style.

The central abstraction of the MIDP UI is a displayable, i.e. an object that has the capability of being placed on the display. `Display` class represents the display hardware, whereas `Displayable` is something that can be shown on it. The high-level API is based on `Screen`, i.e. an application window which fills up the whole area of the device display (except for a thin strip at the top or bottom of the display). `Screen` class inherits from `Displayable`, and all high-level API windows are implemented by classes that inherit from the `Screen` class. Only one screen can be visible at a time, and the user can traverse only through the items on that screen, or switch to another screen.

The Display class and all other user interface classes of MIDP are located in the package `javax.microedition.lcdui`. To show something on a MIDP device screen, you need to obtain the device's display. The Display class provides a `setCurrent()` method that sets the current display content of the MIDlet.

When a MIDlet application is first started, there is nothing displayed on the device screen and there is no current `Displayable` object. It is the responsibility of the application to ensure that some `Displayable` is visible and can interact with the user at all times. Therefore, the application should always call setCurrent() as part of its initialization. The display hardware (of a device) can be accessed by calling the static method `getDisplay()`, where the MIDlet itself is given as a parameter. To show a Displayable object we must use the `setCurrent()` method on the Display object, as on example listing below:

```
Form exampleForm = new Form ("Display some Displayable");
Display display = Display.getDisplay(this);
display.setCurrent (exampleForm);
```

In following subsections, we shortly review some descendants of basic Screen class which with great probability will be used in laboratory programs:

Alert

The simplest subclass of a `Screen` is an `Alert`. It consists of a label, text, and an optional Image. It is possible to set a period of time the `Alert` will be displayed before another Screen is shown. The parameter must either be a positive time value in milliseconds, or the special value `FOREVER`. In this second case, the `Alert` will be shown until the user confirms it. The following code creates an `Alert` with the assigned title, content text, and displays it until it is confirmed by clicking "Done" button:

```
Alert exampleAlert = new Alert ("Title of Example Alert");
exampleAlert.setTimeout (Alert.FOREVER);
exampleAlert.setString ("Text displayed in the Alert window");
display.setCurrent (exampleAlert);
```

Other methods of an Alert class enable to set remaining properties:

- `setType(AlertType)` – the type of generated alert (i.e. constant from `AlertType` class: `ALARM`, `ERROR`, `INFO`, etc) which usually influences the generated alert sound,
- `setImage(Image)` – an icon/image displayed in Alert window,
- `setIndicator(Gauge)` – an activity indicator showing the passing time of the `Alert` displaying period.

Forms and Items

The most important subclass of `Screen` is the class `Form`. In many aspects it is analogous to `Forms` known from Standard Edition API. A `Form` can hold any number of items such as `StringItem`, `TextField`, `ChoiceGroup`, etc. Items can be added to the `Form` using the `append()` method. Possible subclasses of `Item` are presented in table 2-1:

Table 2-1 Subclasses of `Item`, which can populate a `Form` (an application window)

Item	Description
`ChoiceGroup`	Enables selection of elements in a group.
`DateField`	Used for editing date and time information.
`Gauge`	Displays a bar graph for integer values.
`ImageItem`	Used to control the layout of an Image.
`StringItem`	Is a simple read-only text element.
`TextField`	Holds a single-line input field.

All the above items have one common property `Label`, which can be managed by `setLabel(String)` and `String getLabel()` methods. The `Label` should give the short description of the `Item` meaning. It could be removed by setting to `null`, but its presence and descriptiveness is very important in the case of interactive items like `TextField`, values of which are usually entered in separate full screen window entitled with the item's `Label`. Without the labels, it is difficult to associate the sense of entered values.

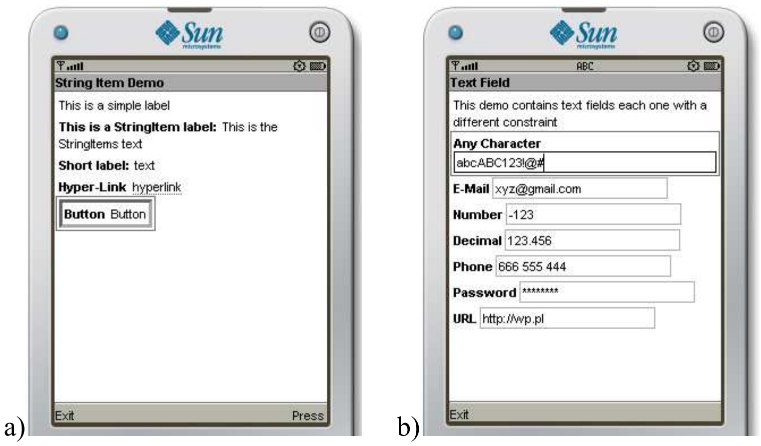
In contrast to the Abstract Windows Toolkit, the `Item` components cannot be positioned or nested freely. `Item`'s method `setLayout(int layout)` enables to influence the relative position of `Item` to positions of other items in sequence, where layout is a composition of layout constants combined using the bit-wise OR operator '|'. Example of layout constants are:

- `LAYOUT_LEFT`, `LAYOUT_RIGHT`, `LAYOUT_CENTER` – values indicating horizontal alignment
- `LAYOUT_TOP`, `LAYOUT_BOTTOM`, `LAYOUT_VCENTER` – values indicating vertical alignment
- `LAYOUT_NEWLINE_BEFORE`, `LAYOUT_NEWLINE_AFTER` – constants setting line breaking points in the stream of successive items.

Thorough description of all components of MIDlet user interface API is provided in JavaDoc documentation of `javax.microedition.lcdui` package. A very good illustration of every item functionality and appearance, is a User Interface Demo application which can be found in <u>File→New Project→Samples→Java ME</u> project category entitled as "UI Demo". In this place, we describe only two `Item` components: `StringItem` and `TextField`, which certainly will be used in student's applications created in the course of this Mobile Computing laboratory.

StringItem

A `StringItem` is a simple display-only component with three characteristic properties: label, textual content and appearance mode. Both the label and the textual content of a `StringItem` may be modified by the application at any moment by calling `setText(String)` and `setLabel(String)` methods, but cannot be directly edited by the user. The appearance mode can be set at `StringItem` creation phase as a third argument of the constructor. The appearance could be one of the constants: `Item.PLAIN`, `Item.HYPERLINK`, or `Item.BUTTON` and influence the way how the `StringItem` text is displayed on the screen (as a plain text, like a hyperlink in a browser or as a push button). Listing 2-3 illustrates Java code which could be used to create StringItems collection presented on figure 2-5.a.



String Item Demo

This is a simple label

This is a StringItem label: This is the StringItems text

Short label: text

Hyper-Link hyperlink

Button Button

Text Field

This demo contains text fields each one with a different constraint

Any Character abcABC123!@#

E-Mail xyz@gmail.com

Number -123

Decimal 123.456

Phone 666 555 444

Password ********

URL http://wp.pl

Figure 2-5 Illustration of `StringItem` and `TextField` appearance in "UI Demo"
a) `StringItems` with different `appearanceMode` (`PLAIN`, `HYPERLINK`, `BUTTON`)
b) `TextFields` with different `constraints` (`ANY`, `EMAILADDR`, `NUMERIC`, etc)

Listing 2-3 Snippet of StringItem Demo application code.

```
  Form mainForm = new Form("String Item Demo");

  mainForm.append("This is a simple label");

  mainForm.append( new StringItem( "This is a StringItem label: ",
                                   "This is the StringItems text" ));

  mainForm.append( new StringItem("Short label: ", "text") );

  mainForm.append( new StringItem("Hyper-Link ", "hyperlink",
                                   Item.HYPERLINK) );

  mainForm.append( new StringItem("Button ", "Button", Item.BUTTON) );
```

TextField

Textual input is handled by the class `TextField`. In MIDP emulator, text can be entered directly into a TextField either by clicking the number buttons in the emulator or by typing on the keyboard. However, majority of MIDP real phone implementations show a separate screen for edition of each. The constructor of TextField takes four values: a label, initial text, a maximum text size, and constraints:

```
TextField(String label, String text, int maxSize, int constraints);
```

The different constraints allow the application to request that the user's input be restricted in a variety of ways. For example, if the application requests the NUMERIC constraint, the implementation must allow only numeric characters to be entered. A basic restrictive constraint settings are listed in table 2-2:

Table 2-2 Graphics constants used to restrict user input to TextField instances

Constant	**Meaning**
ANY	Allows any text to be added.
EMAILADDR	Adds a valid e-mail address, for instance me@mail.com
NUMERIC	Allows integer values.
PASSWORD	Lets the user enter a password. Entered text is masked with '*'.
PHONENUMBER	Lets the user enter a phone number.
URL	Allows a valid URL.

Figure 2-5.b illustrates possible use of these constraints in a Form containing several different text field inputs.

Event Handling with Commands and Listeners

Receiving changes and events generated by high-level screens and items in MIDP is based on a listener model, similar to standard edition AWT API. There are two kinds of listeners: CommandListener and ItemStateListener. The former, is related to a very flexible user interface concept, a Command. A Command is something the user can invoke. It could be GUI button, hardware button, text menu option, voice recognized command, etc. Usually Commands are implemented by so-called *soft buttons*, i.e. additional hardware buttons without fixed functionality. Typical phone device provides at least two such buttons placed just below phone display. Their functionalities are dynamically assigned by textually displayed button "names" (on the bottom strip of the display) depending on the application context. The device determines how the commands are shown on the screen. If the number of activated Commands exceeds the number of available hardware buttons, a separate text *menu* is created automatically to collect all exceeding commands. To create a Command, we need to supply a label, a type, and a priority:

```
Command(String label, int commandType, int priority);
```

The label string contains the displayed name of the button/option. The commandType, provided by the developer during Command construction, is an additional hint for the device system, about where and how to arrange the displaying of the Command. It is used to signify commonly used Commands and place them in the manner most expected by the user. For example, if in the native phone system, the exit operation is always assigned to the leftmost *soft button*, the MIDP will make the same assignment for the Command.EXIT command. Example command types are listed in table 2-3:

Table 2-3 `Graphics` constants used to indicate a `Command` type

Command type	**Meaning**
`BACK`	Returns to the previous screen.
`CANCEL`	Standard negative answer to a dialog
`EXIT`	For exiting from the application.
`HELP`	A request for on-line help.
`ITEM`	Specific to the `Items` of the `Screen` or the elements of a `Choice`
`OK`	Standard positive answer to a dialog
`SCREEN`	An application-defined command
`STOP`	A command that will stop some currently running process, operation, etc.

Every `Command` has a priority. Lower numbers indicate higher priority. The priority '0' `Command` will most probably be shown up on the screen directly (as the *soft button*). The other `Commands` will most likely end up in a secondary *menu*. To create a standard `OK` command, for example, we would do this:

```
   Command commandOK = new Command("OK", Command.OK, 0 );
```

Created commands could then be added (or removed) to a `Form` or any other subclass of `Displayable` by using the following `Displayable` inherited methods:

```
   public void addCommand( Command newCommandObject );
   public void removeCommand( Command removedCommandObject);
```

Added commands are shown on the related `Displayable` and generate appropriate events when a user invokes them (by pressing a button or selecting a menu option). As we stated before, the MIDP uses a classic Java listener model for detecting command actions. The events are sent to the `commandAction` callback method of the associated `CommandListener`. At least one of the MIDlet objects (or the MIDlet itself) has to implement `CommandListener` interface, and should be registered to the considered `Displayable` using the `setCommandListener(listener)` method. Only one listener is allowed for each `Displayable` class instance, but the same listener can be registered to several displayables. The `commandAction()` method receives two parameters:

```
  public void commandAction(Command comm, Displayable disp);
```

The `comm` parameter object identifies the `Command` which was invoked. The `disp` parameter identifies the `Displayable` on which this event has occurred. Using these parameters the `commandAction` implementation can distinguish different sources of the event and generate an adequate action on adequate `Diplayable`. `CommandListener` method should return immediately because MIDP specification does not require the platform to create several threads for the event delivery. Thus, if a `CommandListener` method return is delayed, the system may be blocked. Listing 2-4 illustrates single `Form` MIDlet with two associated `Commands`.

Listing 2-4 Example implementation of Hello World start-up MIDlet

```
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;

public class HelloWorld extends MIDlet implements CommandListener {

  private Form mainForm;

  public HelloWorld () {
    mainForm = new Form("Hello World");
    mainForm.append(new StringItem(null, "Welcome!"));
    mainForm.addCommand(new Command("Exit", Command.EXIT, 0));
    mainForm.addCommand(new Command("Change item", Command.ITEM, 1));
    mainForm.setCommandListener(this);
  }

  public void startApp() {
    Display.getDisplay(this).setCurrent(mainForm);
  }

  public void pauseApp() {}

  public void destroyApp(boolean unconditional) {}

  public void commandAction(Command c, Displayable s) {
    if( c.getCommandType() == Command.EXIT )
      notifyDestroyed();
    else
      ((StringItem)mainForm.get(0)).setText("Press Exit! Goodbye!");
  }
}
```

Summing up, `CommandListener` with it's `commandAction` method is dedicated to handle signals from *softbuttons* and *menu* elements.

The second interface `ItemStateListener` is dedicated to receive events that indicate changes in the internal state of the interactive `Items` within a `Form` screen (value adjustments of an interactive `Gauge`, `TextField` value changes, switching the state of `ChoiceGroup`, etc.) . Most of items in a `Form` can fire events when the user changes them. Our application can listen for these events by registering an `ItemStateListener` (with the method: `setItemStateListener (ItemStateListener iListener)`). When an item state change is detected, the listeners method `itemStateChanged(Item item)` is called.

Exercise J2ME.ex3
Drawing Low Level Graphics on the Device Screen

Create a MIDlet application utilizing Canvas object for implementing custom user interface composed of geometrical figures and animation.

a) Implement selected two-dimensional computer game (like Ping-Pong, PacMan or Arkanoids) which demands to perform some simple drawing and animation.

b) For devices with a touch screen, supporting pointer events, create a simple "Paint"-like graphical editor, which allows one to draw basic figures (lines, rectangles, circles)

MIDP profile offers a low level API based on `Canvas` class from package `javax.microedition.lcdui`, that allow us fully control the device's screen at pixel level. `Canvas` is another subclass of `Displayable`, so in result, a MIDlet can mix and match regular full screen Displayable (like `Forms`, `Alerts`, `Lists`, etc) with full screen graphics `Canvases`. For example, a game application can be composed of high-level `Forms` (to modify game setup, configuration or player preferences in the text-based mode) and several `Canvases` representing different graphic scenes of the game course. But in contrast to AWT, `Canvas` graphics does not allow to mix high-level and low-level components on the same screen simultaneously.

The Canvas provides a `paint(Graphics g)` callback method, similar to the `paint()` method in custom AWT or Swing components. Whenever the MIDP system determines that it is necessary to redraw the content of the device screen, the `paint()` callback method of `Canvas` is called. A typical Canvas implementation is presented on Listing 2-5:

Listing 2-5 Simple demo of typical Canvas implementation

```
import javax.microedition.lcdui.*;

public class DrawingDemoCanvas extends Canvas {
   public void paint(Graphics g) {
      // Draw stuff using Graphics object <g>
      . . .
      g.setColor(128,255,0);
      g.drawLine(0,0,10,20);
      . . .
   }
}
```

In order to see our DrawingDemoCanvas on the screen, we need to set it as a current Displayable object for the MIDlet device display. Example of such operation is presented on Listing 2-6.

Listing 2-6 Setting example Canvas as a current Displayable

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class DrawingDemoMIDlet extends MIDlet {
   public void startApp(){
      Display.getDisplay(this).setCurrent(new DrawingDemoCanvas());
   }

   public void pauseApp(){ }
   public void destroyApp(boolean unconditional){ }
}
```

Using the `Canvas` object, we can draw whatever we want on the screen, find out exactly which keys a user is pressing on, and detect pointer position, if the device is supporting stylus, touch screen or a trackball.

The `Graphics` class, which is an argument of a Canvas `paint()` method, contains methods for drawing shapes, text, and images on the Canvas surface. MIDP's Graphics class is similar to the Graphics and Graphics2D classes in J2SE but it is much smaller. Coordinate space of the drawing is based on the pixels of the device screen. By default, the origin of this coordinate space (0,0) is located in the upper-left corner of the Canvas. We can find out the size of the Canvas by calling `getWidth()` and `getHeight()` methods which return surface dimensions in screen pixels. In the case it could be useful, we can adjust the origin of this coordinate space by calling the `translate(int x, int y)` method, which translates the origin of the graphics context to the point (x, y). To find out the location of the translated origin relative to the default origin, call `getTranslateX()` and `getTranslateY()`.

Table 2-4 Drawing methods of J2ME Canvas

Method name and arguments	**Description**
`drawLine(int x1, int y1, int x2, int y2)`	Draws a line from point (x1,y1) to (x2,y2)
`drawRect(int x, int y, int width, int height)`	Draws an empty rectangle with the upper-left corner at the given (x,y) coordinates, with the given width and height
`drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)`	Like drawRect(), plus additional radii are given for rounded corners of rectangle
`drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)`	Draw the outline of a circular or elliptical arc covering the specified rectangle and angles
`fillTriangle(int x1, int y1, int x2, int y2, int x3, int y3)`	Fills the triangle specified by three corners (x1,y1), (x2,y2), (x3,y3)
`fillRect(int x, int y, int width, int height)`	Fills the area of the rectangle (x,y) (x+w,y+h)
`fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)`	Fills the specified rounded corner rectangle with the current color.
`fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)`	Like drawArc(), but fills the corresponding region with the current color
`drawImage(Image image, int x, int y, int align)`	Draws the specified image aligned relative to the specified anchor point (x,y)

All basic drawing methods of the `Canvas` are listed in the Table 2-4. First four methods draw only outline of a figure (using the concept of current color and line style), the following four methods draw shapes filled with the current color. The last primitive, draws an image which was earlier created in memory, loaded from application resources or downloaded from network. Alignment for the `drawImage()` method is created by composing vertical alignment constants (`Graphics.TOP`, `VCENTER`, `BOTTOM`) with the horizontal alignment constant (`Graphics.LEFT`, `CENTER`, `RIGHT`).

For grayscale devices `Graphics` provides `setGrayScale(int greyness)` as a method to control figure shade. An argument is a number from 0 (for black) to 255 (for white). Current grayscale value can be retrieved by calling `getGrayScale()`.

In devices with color screen, colors are represented as combinations of red, green, and blue, with 8 bits for each color component. We can set the current drawing color using the method `setColor(int RGB)`, where the color argument is specified in the form of 0x00RRGGBB (least significant eight bits giving the blue component, the next eight the green, and the third eight bits giving the red component, the highest order byte is ignored). An alternate three argument method `setColor(int red,int green,int blue)` accepts red, green, and blue values as integers in the range from 0 to 255.

Different devices can support different levels of color representation. Starting from monochrome (black & white) known as "1-bit color", up to full 24-bit color. Two `Display` methods `isColor()` and `numColors()` return useful information about the color capabilities of the device:

```
boolean Display.getDisplay(this).isColor();    //monochrome or color
int     Display.getDisplay(this).numColors();  //number of colors
```

Example Listing 2-7 illustrates sequence of Canvas drawings to create smiling face

```
Listing 2-7 Example Canvas paint() implementation – a smiling face

 public void paint(Graphics g) {
    // Calculate face size in relation to screen dimensions
    int w = getWidth();
    int h = getHeight();
    int size = Math.min ( w, h ) / 2;

    // Clear the background and translate the graphics origin
    g.setColor ( 255, 255, 255 );
    g.fillRect ( 0, 0, w, h );
    g.translate ( w/2, h/2 );

    // Draw the face filling and outline
    g.setColor ( 255, 255, 200 );
    g.fillArc ( -size/2, -size/2, size, size, 0, 360 );
    g.setColor ( 0, 0, 0 );
    g.drawArc ( -size/2, -size/2, size, size, 0, 360 );

    // Draw mouth and eyes
    g.drawArc ( -size/6, size/7, size/3, size/6, 0, -180 );
    g.setColor ( 150, 150, 255 );
    g.fillArc ( -size/6-10, -size/6-10, 20, 20, 0, 360 );
    g.fillArc ( +size/6-10, -size/6-10, 20, 20, 0, 360 );
 }
```

Lines can be drawn with two line styles represented by constants in the Graphics class: `Graphics.SOLID` (the default) and `Graphics.DOTTED` for dotted lines. We can set or retrieve the current style using methods: setStrokeStyle() and getStrokeStyle().

Apart from geometrical figures, graphic interfaces usually contain texts written with different decorative fonts and sizes. Canvas graphics provide three methods for drawing text specified as an ASCII code, an array of chars or a String:

```
public void  drawChar(char character, int x, int y, int anchor);
public void drawChars(char[] data, int offset, int length,
                       int x, int y, int anchor);
public void drawString(String str, int x, int y, int anchor);
```

Assigned text is drawn relative to the location and type of anchor. Anchor location is specified with coordinates (x,y). Anchor type is defined by horizontal (Graphics.LEFT, HCENTER, RIGHT) and vertical (Graphics.BOTTOM, BASELINE, TOP) components. The following example shows how to locate text at various areas/corners of the Canvas (Listing 2-8)

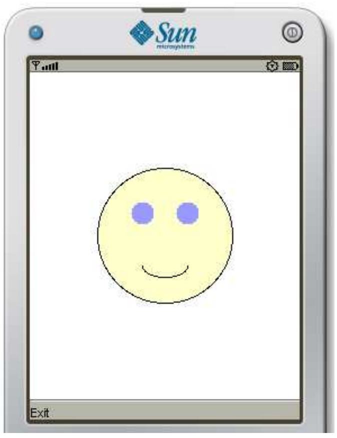
Listing 2-8 Demo of drawing text (**drawString**) on Canvas

```
import javax.microedition.lcdui.*;

public class TextDemoCanvas extends Canvas {
   public void paint(Graphics g) {
   int w = getWidth();
   int h = getHeight();
   g.setColor(0xffffff);
   g.fillRect(0, 0, w, h);
   // Label the four corners
   g.setColor(255,0,0);
   g.drawString("top-left", 0, 0,Graphics.TOP|Graphics.LEFT);
   g.drawString("top-right", w, 0,Graphics.TOP|Graphics.RIGHT);
   g.setColor(0,255,0);
   g.drawString("bottom-left", 0, h,Graphics.BOTTOM|Graphics.LEFT);
   g.drawString("bottom-right", w, h,Graphics.BOTTOM|Graphics.RIGHT);
   // Mark the center of the screen
   g.setColor(0,0,255);
   g.drawString("center",w/2,h/2,Graphics.BASELINE|Graphics.HCENTER);
   }
}
```

It is also possible to control: face, style and size of drawn fonts. There are three available font faces represented by constants in the Font class: FACE_SYSTEM, FACE_MONOSPACE, and FACE_PROPORTIONAL. Font style can be combined by binary "or" operation of STYLE_PLAIN, STYLE_BOLD, STYLE_ITALIC, and STYLE_UNDERLINE constants. The size can be simply SIZE_SMALL, SIZE_MEDIUM, or SIZE_LARGE.

For example, to create a large, bold, italic, proportional font and to set it active style for subsequent text drawings, the following two calls should be done:

```
   Font myFont = Font.getFont( Font.FACE_PROPORTIONAL,
                       Font.STYLE_ITALIC | Font.STYLE_BOLD,
                       Font.SIZE_LARGE);
   graphics.setFont(myFont);
```

a) 

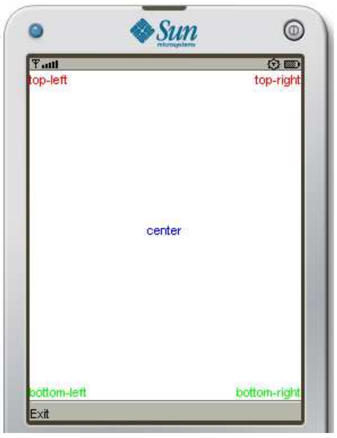
b) 

Figure 2-6 Drawing results of paint() implementation from listings a) 2-7 b) 2-8

Because the `Canvas` class inherits from `Displayable`, it provides the same support for `Command` (soft buttons) as classes from the high-level interface (e.g. `Form` or `Alert`). To arrange user input interactions with `Canvas` contents, standard approach with `CommandListener` interface could be used. But, `Canvas` also handles events at a lower level, which enables to handle interaction with the individual keys of a device. There are three callback methods which are called whenever the user presses and releases a key:

```
   protected void keyPressed(int keyCode);
   protected void keyReleased(int keyCode);
   protected void keyRepeated(int keyCode);
```

All three methods provide an integer argument denoting the Unicode character code assigned to the manipulated button on the device keyboard. If a button has no Unicode correspondence, the given `keyCode` has negative value. Because most MIDP devices have phone keypad, `Canvas` provides several constants corresponding to standard keypad numeric keys from `KEY_NUM0`, `KEY_NUM1`, `KEY_NUM2`, ..., to `KEY_NUM9`, and including `KEY_STAR` and `KEY_POUND`. Devices may have more keys than these (pressing of which will result in device-specific key codes), but portable applications should not rely on the presence of any additional key codes. If it is necessary, an application can get the real *name* of the pressed key by calling `getKeyName(key)` method.

Nowadays, most of mobile devices support a pointing device like stylus and a touch-sensitive screen. The `Canvas` API provides methods for testing pointer device support: `hasPointerEvents()` and `hasPointerMotionEvents()`. If the device supports pointer events, the following callback methods get called when the pointer is pressed, released or dragged:

```
   protected void pointerPressed(int x, int y);
   protected void pointerReleased(int x, int y);
   protected void pointerDraged(int x, int y);
```

While implementing any of abovementioned user input event callback methods, please remember, that all of them and the `paint()` method are called serially. All these methods should return quickly to protect against situation when the user interface becomes frozen. Any longer processing should be served in a separate thread.

In the case of programming real-time graphic applications like computer games or animations, several additional issues should be taken into account. The first one is repainting the `Canvas`. If we want to refresh the display content, we can not call `paint()` method directly (because we can't provide the appropriate `Graphics` argument). Instead, the MIDlet should call `Canvas.repaint()` method, which notifies the AMS system that a repaint is necessary. Unfortunately, the call of `paint()` is not performed immediately. It may be delayed, until the control returns from other event handling methods. The AMS system can even merge several repaint requests. To coerce the implementation to service all the repaint requests without delay, the method `serviceRepaints()` should be used. `Canvas` does not automatically clear itself when we call `repaint()`. The application should clear the screen in the `paint()` method .

By default, `Canvas` supports a full-screen mode. Some devices can supports an alternate modes for `Canvas`, where top or bottom strip is reserved to present information about the state of the device. To force switching into real full-mode, we can use the method `Canvas.setFullScreenMode(true)`.

All MIDlet `Displayables`, in some situations can go to background (hide) and after some time get back to the foreground (become shown). Each time the `Canvas` is shown, the `showNotify()` method will be called. If another `Displayable` is shown, or the application manager switches to run a different application, `hideNotify()` is called. The game interaction can be additionally made more attractive with `Display` methods: `vibrate(int duration)`, `flashBacklight(int duration)`, which enable to control device body vibration and the screen backlight (duration is given in milliseconds).

Students who are more interested in computer games can extend this third exercise to utilize `javax.microedition.lcdui.game` optional package or JSR184, the optional API for rendering of 3D graphics. The `Game` package contains game API, a set of classes that simplify development of two-dimensional games. It provides game-specific capabilities such as an off-screen graphics buffer, the ability to query key status (useful for detecting user input in a game thread) and supports `LayerManager` which can combine multiple layers to create complex scenes. The `Sprite` class supports heroes animation and collision detection. Large scenes or backgrounds can be constructed efficiently using `TiledLayer`.

Using Mobile 3D Graphics (JSR 184), it is possible to render 3D graphics to a 2D MIDP component such as a `Canvas` by binding to the singleton `Graphics3D` instance. `Graphics3D` controls lights for illuminating a 3D scene and camera to determine the portion of the scene that is displayed. All the objects inside the animation world can be located and manipulated using immediate mode API calls.

Exercise J2ME.ex4 Wireless Messaging by SMS Texting

Create MIDlet applications communicating with each other through SMS-based communication channel.

a) Create a distributed implementation of tic-tac-toe (noughts and crosses) game which enables to play the game remotely between two SMS subscribers.
b) Create an application which automates the process of SMS-based broadcasting of selected predefined text messages, to predefined group of phone subscribers (for example: inviting friends to a Friday party, informing superiors about illness or absence)

Using the Internet on the phone could be quite costly because usually connection time is billed per minute or per packet. An interesting lower cost alternative is the Short Message Service (SMS), which is one of the most widely available and popular services for cell phones. The Wireless Messaging API (WMA 1.1, JSR-120), is an optional API that enables MIDP applications to send SMS messages. WMA version 2.0 (JSR-205), adds the support for Multimedia Message Service (MMS) with the ability to send large binary messages consisting of multiple parts. These messages could carry: images, sound, video, or multimedia presentations. There are many possible applications of WMA: chat-type applications, interactive gaming, event reminders, e-mail notification, informational services, etc.

All the interfaces and classes in WMA are a part of the package `javax.wireless.messaging`. WMA is built on top of CLDC and Generic Connection Framework, so we get a `MessageConnection` by passing in an address to the `connect()` method, analogically to `SocketConnection` and `DatagramConnection` in the GCF. But in contrast to the above, we cannot open an input or output stream from it. MessageConnections are only used to send and receive messages. The SMS URL address has to have the format: "sms://" + phone_number + ":" + port_number. Second part, with port_number could be omitted. Examples of URL connection strings are:

- "sms://+111222333" → to send SMS message to the specified phone number
- "sms://+111222333:4444" → to send SMS to a port 4444 on the specified phone
- "sms://:4444" → to open server mode connection on port 4444

Server mode connections are used to receive incoming messages. The concept of a port allows multiple applications to accept messages on the same device. It also enables the device to differentiate between SMS messages destined for a WMA application and standard text messages. The cost of using a port in SMS address is eight bytes of data at the beginning of an SMS message.

To send a single SMS message using WMA, first create an empty message using the `newMessage()` method of the open `MessageConnection` which works as the class factory for messages (Listing 2-9 ①). We also need to specify the message type as `MessageConnection.TEXT_MESSAGE` for SMS text message.

Listing 2-9 Example code sending SMS text message

```
import java.io.*;
import javax.microedition.io.*;
import javax.wireless.messaging.*;

MessageConnection smsConnection;
TextMessage       smsMessage;
. . .
try {
  smsConnection=(MessageConnection)Connector.open("sms://5550000");
  smsMessage=(TextMessage)                                            ①
            smsConnection.newMessage(MessageConnection.TEXT_MESSAGE);
  smsMessage.setPayloadText("text of example SMS");                   ②
  smsConnection.send(smsMessage);                                     ③
} catch (IOException ex) {
  // manage an exception
}
```

Then, set the text string that you want to send ② using `setPayloadText()`. Finally, use the `send()` method of `MessageConnection` to send the message ③.

There are two options to receive SMS messages by a MIDlet. The first one is to use a `MessageConnection.receive()` method. Since `receive()` is a blocking call, it should always be called in its own thread. Listing 2-10 demonstrates how to use this first option.

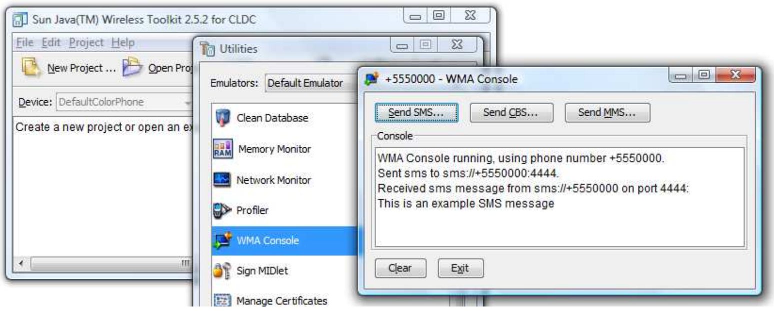
Listing 2-10 Receiving SMS message with blocking method receive()

```
MessageConnection serverConnection;
Message           receivedMessage;

try {
  serverConnection=(MessageConnection)Connector.open("sms://:4444");
  receivedMessage=serverConnection.receive();
  if( receivedMessage instanceof TextMessage ) {                      ①
    String smsText =((TextMessage)receivedMessage).getPayloadText();
    String senderAddress = receivedMessage.getAddress();
    Date   messageDate = msg.getTimestamp();                          ②
    // manage received smsText
  }
} catch (IOException ex) {
  // manage an exception
}
```

Because received message can be of `TextMessage` or `BinaryMessage` type, we have to check it before retrieving message payload ①. In addition to the text payload, two other interesting informations appear in the message header, which can be retrieved by `getAddress()` and `getTimestamp()` methods ②.

MessageConnection also supports a second, nonblocking (event listener-based) way for receiving SMS messages. To use this, we need to create and register a `MessageListener` object with the `setMessageListener()` method, and handle the callback on the `notifyIncomingMessage()` method of this `MessageListener` interface. Only one listener can be registered at a time for the `serverConnection`. The callback is performed by WMA on a system thread, so the `notifyIncomingMessage()` method should return as soon as possible, and any laborious message processing must be performed in a separate thread.

To ease testing of MIDlet SMS applications on debugger, we can use a Sun Wireless Toolkit (WTK) utility called the WMA console, which monitors wireless messaging activities and enable to simulate sending trial SMS/CBS/MMS messages. WMA console can be launched, by starting:
C:\Program Files\NetBeans 6.5.1\ mobility8\WTK2.5.2\bin\ktoolbar.exe
and then selecting File→Utilities→WMA console from Wireless Toolkit main menu.



```
WMA Console running, using phone number +5550000.
Sent sms to sms://+5550000:4444.
Received sms message from sms://+5550000 on port 4444:
This is an example SMS message
```

Figure 2-7 Utilizing WMA console to monitor SMS messaging activities of phone emulator

The second way of testing WMA applications is to use two or more running instances of the device emulator. Each of the instances has its own unique phone number displayed on the device window title bar (default are: +5550000, +5550001, etc.). If sending messages between these numbers, the emulator environment will take care of simulating wireless transmission between emulated devices.

The Wireless Messaging API, extends the CLDC's Generic Connection Framework, adding the ability for J2ME applications to send and receive SMS messages. The latest JSR 205, specifying WMA 2.0, extends this further to take advantage of the content-rich Multimedia Message Service (MMS) that is becoming available starting from 2.5G and 3G phone networks. Presented exercise, can be extended to enable transmission of `MultipartMessage` carrying an image, an audio or video clip file. A `MultipartMessage` is a subinterface of `Message`. Apart from basic methods like `setAddress()` inherited from `Message`, additional methods on this interface deal with the additional features owned by MMS messages (for example: `addAddress(String type, String address)` to set multiple recipients for "to", "cc", and "bcc" fields similarly to e-mail). Listing 2-11 illustrates how to send multipart message containing a png image.

Listing 2-11 Sending *.png image by multipart MMS message

```
//prepare an image buffer with multimedia content (image) to send
String imageName = "/images/example_image.png";
InputStream inputStream = getClass().getResourceAsStream(imageName);
byte[] imageBuffer = new byte[inputStream.available()];
inputStream.read(imageBuffer);
//compose a multipart message
String receiver = "mms://+5550000:5555";
MessageConnection mmsConnection;
MultipartMessage  mmsMessage;mmsConnection = (MessageConnection)
Connector.open(receiver);
mmsMessage = (MultipartMessage)
       mmsConnection.newMessage(MessageConnection.MULTIPART_MESSAGE);
mmsMessage.setSubject("MMS with example png image");
mmsMessage.addMessagePart(new MessagePart(imageBuffer,0,
             imageBuffer.length, "image/png","id1","location", null));
//send the prepared MMS message
mmsConnection.send(mmsMessage);
```

Exercise J2ME.ex5
MIDP Persistent Data Storage Using RMS

Create MIDlet applications that permanently memorize data collected during subsequent sessions or share the same data between several MIDlets (from different Suites).

a) Implement a MIDlet with history-feature, that stores all the Data/Time moments of its launching.

b) Create a composition of three cooperating MIDlets, which share the same RecordStore. At least one of them should be from different MIDlet Suite. Consider three functionalities: record owner, reader (read only) and writer.

c) Create a MIDlet implementation of a home private library database representing books, readers and their relation in the form of three tables.

Most of the mobile applications need to store some data in a permanent way (even when the device is switched off). MIDP defines a set of classes for storing and retrieving data called Record Management System (RMS). This mechanism is modeled after a simple record oriented database and enables to persistently store data and retrieve it later. The central concept for persistent storage is the `RecordStore`. Each `RecordStore` consists of a collection of records that will remain persistent across multiple invocations of a MIDlet. The records data are stored in a dedicated memory part of mobile device which is generally called "persistent storage", but the details of how exactly records are stored are specific to the device implementation and are not exposed to a MIDlet.

Any MIDlet suite that plans to use RMS memory, should specify the minimum number of bytes for the data storage it requires, by setting the MIDlet-Data-Size attribute (in application descriptor and the JAR manifest). If not, the device could assume the MIDlet suite does not require any space for persistent data storage. The RMS is not suitable for storing big amounts of data. The MIDP specification dictates that the minimum amount of persistent storage, which has to be provided, is only 8KB.

The javax.microedition.rms.RecordStore class encapsulates all access to persistent storage. It provides methods for accessing and manipulating RecordStores, as well as methods for working with individual records (arrays of bytes). All access methods are static so can be called without an instance.

Every MIDlet in a MIDlet Suite, can access every Record Store created by its Suite members. Since MIDP 2.0, access across suite borders is also possible. In the case of concurrent access it is important to remember that no locking operations are provided in RMS API. RecordStore implementations have to ensure that all individual operations are atomic, synchronous, and serialized so that no corruption occurs with multiple accesses. However, if a MIDlet uses multiple threads to access a record store, it is the MIDlet's responsibility to coordinate this access.

Every instance of RecordStore is identified by a name. Within a MIDlet suite's record stores, the names must be unique. We can acquire a list of all created record stores names with method listRecordStores() returning the array of strings:

```
 String[] names = RecordStore.listRecordStores();
```

To open an existing (and possibly create a new) record store associated with the given MIDlet suite, openRecordStore() method should be used:

```
public static RecordStore openRecordStore(String  recordStoreName,
                                          boolean createIfNecessary)
     throws RecordStoreException,
            RecordStoreFullException,
            RecordStoreNotFoundException
```

If the record store does not exist, the createIfNecessary parameter determines whether a new record store will be created or not. An opened record store can be closed by calling the closeRecordStore() method. And finally, to remove a record store and delete all its contained records, we have to call the static deleteRecordStore() method.

To enable record stores sharing between different MIDlet Suites, MIDP 2.0 introduced RecorStore authorization mode. The default authorization mode is AUTHMODE_PRIVATE, which means that a record store is only accessible from MIDlets in the sam suite that created the record store. Record stores can be shared by changing their authorization mode to AUTHMODE_ANY. In this case, any other MIDlet on the device can access the record store. Shared record stores can be created and opened using an alternate openRecordStore() method with two additional arguments:

```
public static RecordStore openRecordStore(String  recordStoreName,
                                          boolean createIfNecessary,
                                          byte    authMode,
                                          boolean writable);
```

We can also change the authorization mode and writable flag of an open record store using the setMode() method:

```
public void setMode(byte authmode, boolean writable);
```

Only the owning MIDlet suite can change the mode of a RecordStore. SecurityException will be thrown if the MIDlet Suite is not allowed to change the mode of the RecordStore. Other MIDlets (which are not creators of the RecordStore) can access a shared record with third version of openRecordStore() method:

```
public static RecordStore openRecordStore(String  recordStoreName,
                                          String  vendorName,
                                          String  suiteName);
```

In this case, the MIDlet has to know not only the RecordStore name, but also the name of the MIDlet suite that created it, and the name of the MIDlet suite's vendor. These names must be the same as MIDlet-Name and MIDlet-Vendor attributes in the MIDlet suite JAR manifest file.

Each record in a RecordStore is simply an array of bytes and each of them has an positive integer identification number which starts from 1. To add a new record, we have to supply the byte array to the addRecord() method and returned result is the new record's integer ID:

```
public int addRecord(byte[] data, int offset, int numBytes);
```

The added record will consist of numBytes number of bytes taken from the data array, starting form offset position in this array. The following code snippet (Listing 2-12) illustrates adding a new record to a RecordStore named "MyFriends" which collects the names of somebody's friends. It starts with openinig (or creating) suitable RecordStore, then converts newFriendName from String raw byte array and finally adds the entire array into a new record:

Listing 2-12 Adding of a single String (text) to RMS RecordStore

```
   RecordStore rs = null;
   try {
      rs = RecordStore.openRecordStore("MyFriends", true);
      String newFriendName = "John Smith";
      byte[] rawByteArray = newFriendName.getBytes();
      rs.addRecord(rawByteArray, 0, rawByteArray.length);
   } finally {
      if (rs != null)
         rs.closeRecordStore();
   }
```

Usually we want to store custom objects which have more complicated structure than a single text or number. Because each record has the form of single byte array, we have to begin with serialization of memorized data and only afterwards write the resulting array to a new record. In the case of several different values which do not form a single class, the most comfortable serialization method is to use ByteArrayOutput Stream. Reverse process is needed to recreate the objects (with ByteArray InputStream for deserialization). Listing 2-13 illustrates such serialization process for memorizing three values describing the victory in computer game: score, playerName, victoryDate.

Listing 2-13 Adding a record of three serialized values

```
private byte[] serializeData(int    score,
                             String playerName,
                             long   victoryDate) throws IOException
{  //auxilary function converting fields into raw byte array
   ByteArrayOutputStream baos = new ByteArrayOutputStream();
   DataOutputStream output = new DataOutputStream(baos);

   output.writeInt(score);
   output.writeUTF(playerName);
   output.writeLong(victoryDate);

   output.close();
   baos.close();
   return baos.toByteArray();
}

public void addScore(RecordStore rs,
                     int score, String playerName, long victoryDate)
{
   //Each score is stored in a separate record,
   //formatted with the score, followed by the player name and date.
   byte[] byteArray = serializeData(score,playerName,victoryDate);

   // Add the array to the record store
   try {
      int recId;  // record ID returned by addRecord, here not used
      recId = rs.addRecord( byteArray, 0, byteArray.length );
   }
   catch (RecordStoreException rse) {
      System.out.println(rse);
      rse.printStackTrace();
   }
}
```

Reverse operation can be done with `getRecord()` method which retrieves a byte array of data for the supplied record ID. There are two versions of `getRecord()` method. First one returns newly created byte array with the size fitted to record content. An alternate version of this method puts the record data into an array buffer that you supply as second parameter of the method:

```
public byte[] getRecord(int recordId)
                                 throws RecordStoreNotOpenException,
                                        InvalidRecordIDException,
                                        RecordStoreException

public int getRecord(int recordId, byte[] buffer, int offset)
```

This second version returns the number of bytes that were copied into your array. It is more efficient in the case of massive readings of `RecordStore` content, because it avoids superfluous repetitive creation of byte array buffer. The drawback is, that the array we supplied could be not large enough to hold the record content (and `ArrayOutOfBoundsException` will be thrown). To control this situation, we can find out the size of a particular record ahead of time by calling `getRecordSize()`:

```
   byte[] fittedArray = new byte[rs.getRecordSize(id)];
   rs.getRecord(id, fittedArray, 0);
   String retrievedString = new String(fittedArray);
```

Replacing the data of an existing record is done by calling the setRecord() method:

```
public void setRecord(int recordId,
                      byte[] newData, int offset, int numBytes)
        throws RecordStoreNotOpenException, InvalidRecordIDException,
               RecordStoreException,        RecordStoreFullException
```

The `RecordStore` keeps an internal counter that it uses to assign record IDs. You can find out what the next record ID will be by calling `getNextRecordID()`. And you can find out how many records exist in the `RecordStore` by calling `getNumRecords()`.

The record can be deleted by passing its ID to `deleteRecord()` method. Unfortunately, record IDs are fixed and the deleted record ID is not used again by incoming records. As a result, after several deletions record IDs sequence will not be continuous and direct use of standard iteration process can be very tedious.

In RMS, the `RecordEnumeration` is introduced to simplify traversing the `RecordStore` content by iteration through a set of existing records. We can find out if there's a next record by calling `hasNextElement()`. If the next record exists, we can retrieve its data by calling the `nextRecord()` method or retrieve its ID by calling `nextRecordId()`. A `RecordStore` method `enumerateRecords()` is typically used to create an enumeration for traversing a set of records in the record store in an optionally specified order:

```
public RecordEnumeration enumerateRecords(RecordFilter      filter,
                                          RecordComparator comparator,
                                          boolean          keepUpdated)
                                throws RecordStoreNotOpenExceptionReturns.
```

By using an optional `RecordFilter`, a subset of the records can be chosen that match the supplied filter. By using an optional `RecordComparator`, the enumerator can index through the records in an order determined by the comparator. Both of them, `filter` and `comparator` can be used for implementing search and sorting capabilities typical for every database system. A skeleton of typical full use of `RecordEnumeration` will contain 5 steps:

```
Listing 2-14  Skeleton of filtered and sorted RMS record enumeration

// ① Open a RecordStore rs
    RecordStore rs = RecordStore.openRecordStore(…);

// ② Create a RecordFilter rf
//    the instance of the class implementing a RecordFilter interface
//    with its  boolean matches(byte[] candidate)  method

// ③ Create a RecordComparator rc
//    the instance implementing a RecordComparator interface
//    with its  int compare(byte[] rec1, byte[] rec2)  method

// ④ Create enumeration agregating above filter and comparator
    RecordEnumeration re = rs.enumerateRecords(rf, rc, false);
```

```
// ⑤ walk straight through the selected records
      while (re.hasNextElement()) {
         byte[] recordBytes = re.nextRecord();
         // Process the retrieved bytes.
         // . . .
}
```

Listing 2-15 illustrates example implementation of record enumeration which traverses the whole content of "StudentNames" `RecordStore` and displays all the names in a IDE console window. In this case `RecordFilter` and `RecordComparator` are not used, so the two first parameters of `enumerateRecords()` method are set to `null`. The last parameter is set to `false`, because we do not forecast any `RecordStore` updates during this iteration.

Listing 2-15 Raw browsing through RecordStore composed of all Strings

```
   RecordStore rs = null;
   RecordEnumeration re = null;

   try {
      rs = RecordStore.openRecordStore( "StudentNames", true );
      re = rs.enumerateRecords( null, null, false );
      while (re.hasNextElement()) {
         byte[] rawBytes = re.nextRecord();
         String studentName = new String( rawBytes );
         System.out.println( studentName );
      }
   } finally {
      if (re != null) re.destroy();
      if (rs != null) rs.closeRecordStore();
   }
```

`RecordEnumeration` enables to move the iteration cursor forward and backward. It can be done by using `hasPreviousElement()`, `previousRecord()`, and `previousRecordId()` methods which work just like their `next` counterparts.

Finaly, to find out the whole number of bytes used by a record store, we can call the public `int getSize()` method on a `RecordStore` instance. We can also find out how much more space is available in RMS "persistent memory" by calling the method: public `int getSizeAvailable()`.

To sum up this exercise, there are three more methods which provide access to MIDlet persistent data. The first one utilizes resource files, which are another form of persistent storage. Resource files can be images, text, or other types of files that are stored in a MIDlet suite JAR. These files are read-only. You can access a resource file as an InputStream by using the `getResourceAsStream()` method in a MIDlet's class. A typical usage looks like this:

```
InputStream in = this.getClass().getResourceAsStream("/image.png");
```

The second method is to benefit from File Connection Optional Package (JSR-75). Modern devices may have slots for optional flash memory cards that can be added (like Secure Data / SD cards, Compact Flash, and Memory Stick). The File Connection Optional Package provides an API that can be used by applications to access a device’s file systems. A device may expose its file systems through this optional API which is contained in the `javax.microedition.io.file` package.

The third method is to utilize the Personal Information Management (PIM) package. Many mobile devices, especially phones, have the ability to maintain lists of phone numbers and names. Some devices also store addresses, e-mails, events, to-do lists, and other personal information. This PIM data is stored in PIM on-device databases. PIM Optional Package (from the same package JSR-75) enables MIDlet applications to access PIM databases of mobile device. Using this optional package our MIDlets can read, add, modify, or delete records contained in contacts, events, and to-do lists. So this database could be also treated as a sort of MIDlet's persistent memory.

3. Programming Microsoft Windows Mobile

The second group of exercises gives us an overview of techniques and tools for developing applications for the Microsoft Windows Mobile platform, using the .NET Compact Framework (CF). Windows Mobile platform is the fourth biggest mobile player in the world, with about 10% of the market share. An important aspect of Windows Mobile OS is its close correlation with other Windows Operating System versions (2K, XP, Vista, 7) and Microsoft Office applications (Outlook, Word, Excel). For many business customers it is easier to invest into familiar consumer brand, which is well known to all office workers, than take a risk of innovations from mobile market newcomers. In US, it is the 3rd most popular smartphone operating system for business use with about 25% share among enterprise users.

3.1. .NET Compact Framework Platform

.NET is a programming framework originally developed by Microsoft for servers and desktop computers with Windows Operating System. The .NET Compact Framework (CF) is a subset of the full .NET Framework. It is adopted specifically for resource-constrained devices, such as palmtops (Personal Digital Assistants) and smart mobile phones. The size of memory footprint was scaled down to about 10% of the full framework. Some components and some functionalities, that are not useful on mobile platform, have been removed. Some extra functionalities (for example: management of additional hardware buttons or cradle synchronization) were added. As a result, CF greatly simplifies the process of creating and deploying applications for mobile devices, while providing almost the same benefits as the .NET Framework and allowing the developer to take full advantage of the capabilities of the device.

During this course, the implementation language for Windows Mobile will be C# – an object-oriented programming language developed by Microsoft especially for .NET framework. Code written in C# is called managed code, it targets the .NET Common Language Infrastructure (CLI) and takes advantage of the built-in structures of the .NET Framework. Using "managed code" means that .NET byte-code is executed on the Common Language Runtime (CLR), in a similar manner to Java code being executed on Java Virtual Machine. A key benefit of managed code is that it supports strong type-safety and handles many common errors that plague native code programmers (bad pointers, memory leaks, etc). It is also possible to use other languages to program Windows Mobile applications, for example Java or C++ native code language. For students interested in C++, Visual Studio IDE subdirectory `Samples\PocketPC\Cpp` contains about one hundred of C++ code examples. But usually native C++ code is much more complicated (for instance, simple `ListView` example requires about six hundreds of lines, while C# implementation requires only a few). For a time-restricted laboratory, C# seams to be the best choice. It is also possible to execute Java Micro Edition programs on Windows mobile device, but in that case, low level access to device hardware is much more restricted.

3.2 Visual Studio Integrated Development Environment

Microsoft Visual Studio IDE (2005 or 2008) will be used as a basic development environment for all applications presented in this chapter. It enables automatic targeting the .NET CF assemblies and includes a graphical drag-and-drop designer to facilitate creation of graphical user-interface. It also contains an emulator than can be used to execute and debug the program without having to transfer it to a mobile device.

Windows Mobile programming is unsupported under freely available Visual Studio EXPRESS edition, so it is important to install commercial: Standard, Professional and larger editions of VS. At the moment, the faculty of Electronics is a member of Microsoft MSDN AA program, so all this software can be downloaded (without charge) by students from Microsoft E-Academy page for EKA/PWr:
http://msdn62.e-academy.com/elms/Storefront/Home.aspx?campus=msdnaa_pd6979

All demonstration codes in this chapter are targeted at Windows Mobile 5.0 and higher. Because Windows Mobile 6.0 was shipped after the release of the abovementioned versions of Visual Studio (2005/2008), downloading and installing "Windows Mobile Professional And Standard Software Development Kit Refresh", has to be done from Microsoft mobility developer page. This SDK refresh adds documentation, sample code, header and library files, emulator images and tools to Visual Studio that let you build applications for Windows Mobile 6. The Visual Studio device emulators are great development tools. However, real device testing and debugging is usually unavoidable. For Windows Mobile 5.0 (and later) - powered devices, we need Microsoft ActiveSync version 4.0 or later, or Windows Mobile Device Center on the Windows Vista/7 operating systems.

3.3 Laboratory Exercises

For newcomers in C# programming and Visual Studio IDE, we start with the very simple "ConsoleHello" application for desktop text console (Listing 3-1).

Listing 3-1 Example of simple C# console application

```
//in file "ConsoleHello.cs"
using System;                                                    ①
public class Hello{
    public static void Main(string[] args){                     ②
        Console.WriteLine("Welcome in C# World");
    }
}
```

The program should be straightforward and readable for students with a background knowledge of any other object-oriented languages like Java. First line ① includes the `System` namespace, which is a part of the .NET Framework class library. Namespace in C# is an equivalent to the "package" concept in Java. The following lines define the class `ConsoleHello` and the static `Main` method ②, which is equivalent to the "main" method of Java or C++, automatically executed when the program is started. To try out the above program, select Tools→Visual Studio Command Prompt of the MS Visual Studio IDE main menu. Change the directory to the location where the file

ConsoleHello.cs is placed. Then type a command csc ConsoleHello.cs to invoke the line C# compiler (csc) and finally type ConsoleHello to see "Welcome in C# Word" greeting, written on the console screen by "ConsoleHello.exe" executable.

Exercise WM.ex1
Building the first application with a Windows Forms based GUI

Apply System.Windows.Form library to create a single Form (dialog) application which can be compiled both for stationary desktop station (with a standard display) and for mobile device with a very compact screen.

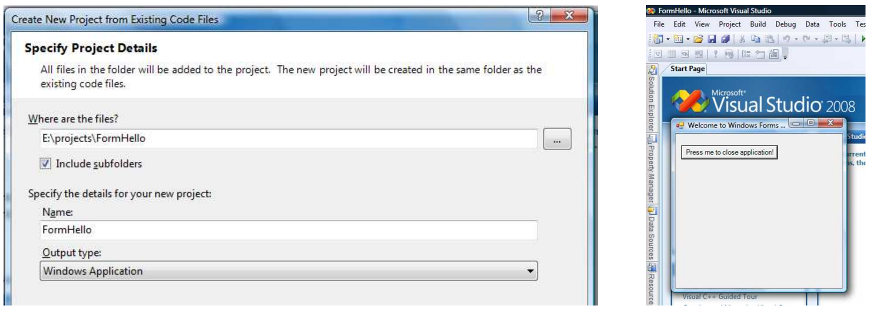
Modern applications usually take advantage of a graphical interface which is much more comfortable for application users, than primitive text-based console applications. To ease development of such software, .NET framework provides standard libraries which contain a large collection of different interface controls like: Form, Button, Label, TextBox, etc. In the following code example (on Listing 3-2), we illustrate the use of the System.Windows.Forms library to create simple graphical application with: one Form, one Button and one button_Click handler.

Listing 3-2 Example of user interface based on Windows.Forms

```
using System;
using System.Windows.Forms;                                          ①
using System.Drawing;

public class FormHello : Form                                        ②
    {
        public static void Main()
        {
            Application.Run(new FormHello());                        ③
        }

        public FormHello()                                           ④
        {
            this.Text = "Welcome to Windows Forms World";
            Button button = new Button();                            ⑤
            button.AutoSize = true;                                  ⑥
            button.Text = "Press me to close this application!";
            button.Location = new Point(10, 20);
            button.Click += new EventHandler(button_Click);          ⑦
            this.Controls.Add(button);
        }

        void button_Click(object sender, EventArgs e)                ⑧
        {
            this.Close();
        }
    }
```

In this example we use three different namespaces ①: basic System namespace (as in the previous application) here for EventHandler, EventArgs, the `System.Windows.Forms` containing `Form` and `Button` classes, and `System.Drawing` for `Point` class. Line ② defines new class `FormHello`, which inherits from the basic `Form` class and contains application starting point: static method `Main`. Line ③ creates and makes visible an instance of our `FormHello` window and starts a new message loop for the application by executing `Application.Run` method. Class constructor ④ sets the main window title (`this.Text`), creates a single instance of `Button` ⑤, sets its label to text "Press me …", and adds method `button_Click` ⑧ as the active `Button.Click` event handler ⑦. In result, clicking the button causes termination of application message loop and application quits.

To try out the above application, save the code in `FormHello.cs` source file. Start Visual Studio and select File→New→Project from existing code option from main menu. Select Visual C# type of project, specify path to the above source file, the name of created project (e.g. FormHello), and set Windows Application as output type. Pay attention, that this time we have selected standard "C# project" and created a standard desktop window application (Fig. 3-1)


Create New Project from Existing Code Files

Specify Project Details

All files in the folder will be added to the project. The new project will be created in the same folder as the existing code files.

Where are the files?: E:\projects\FormHello

☑ Include subfolders

Specify the details for your new project:

Name: FormHello

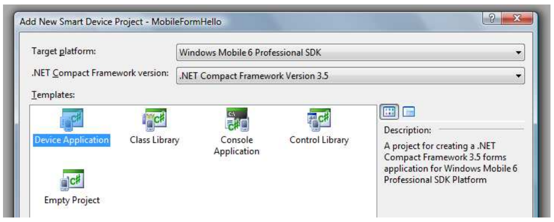
Output type: Windows Application

FormHello - Microsoft Visual Studio
File Edit View Project Build Debug Data Tools
Start Page
Microsoft® Visual Studio 2008
Welcome to Windows Forms ...
Press me to close application!
Solution Explorer
Property Manager
Data Sources
Visual C++ Guided Tour

Figure 3-1 Creating desktop Windows.Forms application from example source code

To debug the above code as a MOBILE application on a mobile device emulator:

1. Create a new project by selecting: File→New→Project→Visual C# →Smart Device as the project type, Smart Device Project as a project template, set the location directory for the code, and give it any meaningful name, for example "MobileFormHello"
2. Choose Device Application template, mobile Target platform SDK and Compact Framework version as shown in figure 3-2


Add New Smart Device Project - MobileFormHello

Target platform: Windows Mobile 6 Professional SDK

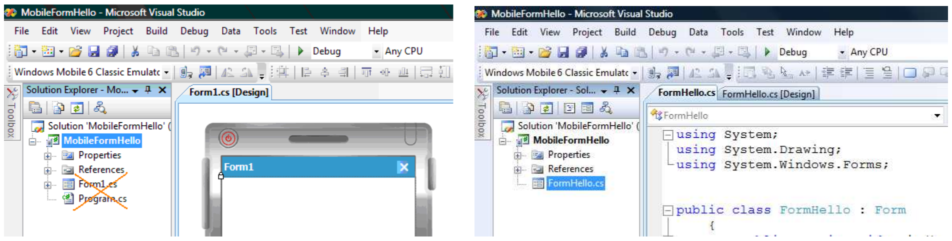
.NET Compact Framework version: .NET Compact Framework Version 3.5

Templates: Device Application, Class Library, Console Application, Control Library, Empty Project

Description: A project for creating a .NET Compact Framework 3.5 forms application for Windows Mobile 6 Professional SDK Platform

Figure 3-2 Creating mobile device application project using WM 6.0 SDK and CF ver. 3.5

3. In Solution Explorer window, delete the default items "Form1.cs", "Program.cs" created by Visual Studio as default setup, and add our "FormHello.cs" file instead of them, using Project→Add Existing Item option from main menu (Fig.3-3). Then try to build and run the application by selecting Debug→Start Debugging


MobileFormHello - Microsoft Visual Studio
File Edit View Project Build Debug Data Tools Test Window Help
Debug
Any CPU
Windows Mobile 6 Classic Emulato
Solution Explorer - Mo...
Form1.cs [Design]
Solution 'MobileFormHello'
MobileFormHello
Properties
References
Form1.cs
Program.cs
Form1
Toolbox
MobileFormHello - Microsoft Visual Studio
Solution Explorer - Sol...
FormHello.cs
FormHello.cs [Design]
FormHello
FormHello.cs

```
using System;
using System.Drawing;
using System.Windows.Forms;

public class FormHello : Form
    {
```

Figure 3-3 Creating Smart Device project type from existing code "FormHello.cs"

At the first attempt to build the application, the C# compiler will signal the error "error CS1061", namely: 'System.Windows.Forms.Button' does not contain a definition for 'AutoSize'. This error illustrates the fact, that .NET Compact Framework omits some parts of the full Framework. The `AutoSize` property of `Button` class is one example of omitted items. The most primitive solution is to comment or delete the line ⑥ setting the `AutoSize` property, but in this case the default button surface area will be too small to display the whole text on the button (see figure 3-4.a). Part of the button label will be cut off. Adding the line which sets the `Button.size` property will solve the problem more correctly (Fig. 3-4.b):

```
//button.AutoSize = true;                                        ⑥
button.Size = new Size(200, 40);
```

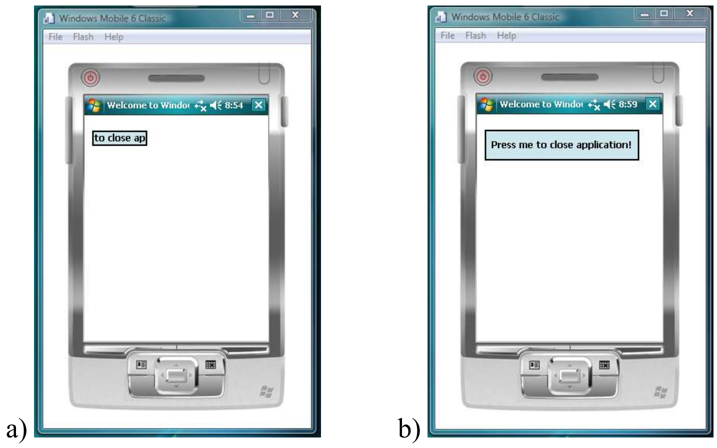


Figure 3-4 Example screenshots of form "FormHello.cs" tested on mobile device emulator
a) without auto-size operation on the button b) with manually corrected `button.Size`

To create a UI for a mobile device application, we can generally follow the same basic process we use to create a UI for any desktop application. However, several "mobile designing" hints should be taken into account during this process. Mobile devices have relatively small screens and designing an effective user interface usually appears a big challenge for mobile developer. Handling and navigating forms in different screen orientations and resolutions, handling user input from touch screen, phone keypad, on-screen keyboard (the Software-based Input Panel - SIP) or full Qwerty hardware keyboard makes it even more difficult.

An obvious UI design guideline, for devices with a touch screen, is to place near the bottom of the screen items that require the user to tap. Otherwise, the user's hand does obscure the screen when the user taps to make a selection. Because of that, the tabs of the TabControl and the main menu bar appear at the bottom of the screen (opposite to desktop applications).

Entering larger portions of text data with the phone keypad or SIP is very tedious for a user, so we should try to minimize the amount of data a user must enter wherever possible. It is suggested to replace text boxes, with alternative controls such as combo boxes, check boxes, and radio buttons, if it is possible. In the case the text boxes are unavoidable, the developer should place them at the top of the display, to prevent them from being obscured by software keyboard panel (SIP) when it becomes visible.

Basic mobile windows `Form` component is always full screen and the user can interact with only one `Form` at a time. Each mobile form, by default, is given a `MainMenu` control. If this control contains only one or two menu items, it is used for naming the "soft keys", displayed just above two additional hardware "function" buttons placed under the screen of all Windows Mobile devices. If the menu contains more than two items or if we use earlier versions of Windows Mobile (earlier than 5.0), it is used for standard menu and toolbar.

Typically, the top strip of the `Form` contains the application title bar. The device uses the top strip for the device start menu, clock, and signal indicators. If we want to remove this strip, `WindowsState` property of the form should be set to `Maximized` (instead of its default value `Normal`). Additionally, if we delete the menu control from application form, it hides the bottom "menu" strip, and the whole device screen will be accessible as `Form` interior.

Exercise WM.ex2
Exploring Important .NET CF Windows Forms Controls

Make use of Visual Studio Designer to create an example Windows Mobile application demonstrating capabilities of mobile version of `Windows.Forms` library, to create an attractive and communicative graphical user interface.
Suggested subjects for applications are:

a) Scientific calculator
b) E-commerce order form editor
c) Multiple-choice tests e-learning application

The Windows Forms controls available to mobile device projects are a subset of the controls available to desktop projects. Additionally, the members of classes existing on the mobile device platform, usually form a subset of the members we will find in the same class on the desktop platform. For example, Compact Framework implementation of standard `Button` control does not have an `Image` property (to display an icon on the button) and lack the abovementioned `AutoSize` property. Detailed description of all `Windows.Form` components can be found in free online MSDN documentation. Below, we shortly summarize some properties of selected components which are expected to be surely used in the course of this series of exercises.

The most useful is the knowledge about characteristic elements of the `Control` type, which is the base class for all controls (components with visual representation). Three following tables gather together common features: the properties (Table 3-1), methods (Table 3-2) and events (Table 3-3) among all `Control` family members.

Table 3-1 Frequently used, common properties of all `Control` inherited classes

Property name	Property description
`BackColor`	Gets or sets the background color for the control.
`Bottom`	Gets the distance, in pixels, between the bottom edge of the control and the top edge of its container's client area.
`Bounds`	Gets or sets the size and location of the control including its nonclient elements, in pixels, relative to the parent control.
`Capture`	Gets or sets a value indicating whether the control has captured the mouse.
`ContextMenu`	Gets or sets the shortcut menu associated with the control.
`Controls`	Gets the collection of controls contained within the control.
`Enabled`	Gets or sets a value indicating whether the control can respond to user interaction.
`Focused`	Gets a value indicating whether the control has input focus.
`Font`	Gets or sets the font of the text displayed by the control.
`Height`	Gets or sets the height of the control.
`Left`	Gets or sets the distance, in pixels, between the left edge of the control and the left edge of its container's client area.
`Location`	Gets or sets the coordinates of the upper-left corner of the control relative to the upper-left corner of its container.
`Parent`	Gets or sets the parent container of the control.
`Text`	Gets or sets the text associated with this control.
`Visible`	Gets or sets a value indicating whether the control and all its parent controls are displayed.
`Width`	Gets or sets the width of the control.

The use of the above properties can be illustrated with the following code snippet, which: creates a new Button ①; sets its initial `Size`, `Location` and `Text` ②; adds it to the main form `Controls` collection ③ and finally removes it from the screen by switching off its `Visible` property ④.

```
Button button = new Button();                                    ①
button.Location = new Point(10, 20);
button.Size = new Size(200, 40);                                 ②
button.Text = "This is a text displayed on my surface";
mainForm.Controls.Add(button);                                   ③
. . .
button.Text = "And now, I'm going to disappear";
button.Visible = false;                                          ④
```

Table 3-2 Frequently used common methods of all classes inheriting from `Control`

Method Name	Method Description
`BringToFront`	Brings the control to the front of the z-order.
`CreateGraphics`	Creates the Graphics for the control.
`Dispose`	Overloaded. Releases all resources used by the Control.
`Focus`	Sets input focus to the control.
`Hide`	Conceals the control from the user.
`Invalidate`	Overloaded. Invalidates a specific region of the control and causes a paint message to be sent to the control.
`OnClick`	Raises the Click event.
`OnDoubleClick`	Raises the DoubleClick event.
`OnKeyDown`	Raises the KeyDown event.
`OnLostFocus`	Raises the LostFocus event.
`OnMouseDown`	Raises the MouseDown event.
`OnPaint`	Raises the Paint event.
`Refresh`	Forces the control to invalidate its client area and immediately redraw itself and any child controls.
`SelectNext Control`	Activates the next control.
`Show`	Displays the control to the user.
`ToString`	Returns a String containing the name of the Component, if any.

Table 3-3 Frequently used, common events typical for all `Control` inherited classes

Event name	Event description
`Click`	Occurs when the control is clicked.
`DoubleClick`	Occurs when the control is double-clicked.
`EnabledChanged`	Occurs when the Enabled property value has changed.
`GotFocus`	Occurs when the control receives focus.
`KeyDown`	Occurs when a key is pressed while the control has focus.
`LostFocus`	Occurs when the control loses focus.
`MouseDown`	Occurs when the mouse pointer is over the control and a mouse button is pressed.
`MouseMove`	Occurs when the mouse pointer is moved over the control.
`Paint`	Occurs when the control is redrawn.
`Resize`	Occurs when the control is resized.

Most of the controls can be considered as isolated visual objects, but there are three controls which are grouping containers for other controls. First one is the earlier discussed main `Form` control, second is a `Panel`, and the third one is a very popular `TabControl`.

Panel control can be used, for example, to group together multiple `RadioButton` controls. Another use of the `Panel` control is for hiding and showing a group of controls. We can create and place two or more panels on the same `Form`, populate them with various controls, and then toggle the boolean `Visible` property of each `Panel`. This technique can be used to avoid creating multiple forms. Instead of switching between forms, we can switch between different panels on the same form by toggling the visibility properties of the panels.

Listing 3-3 Example use of the TabControl container component

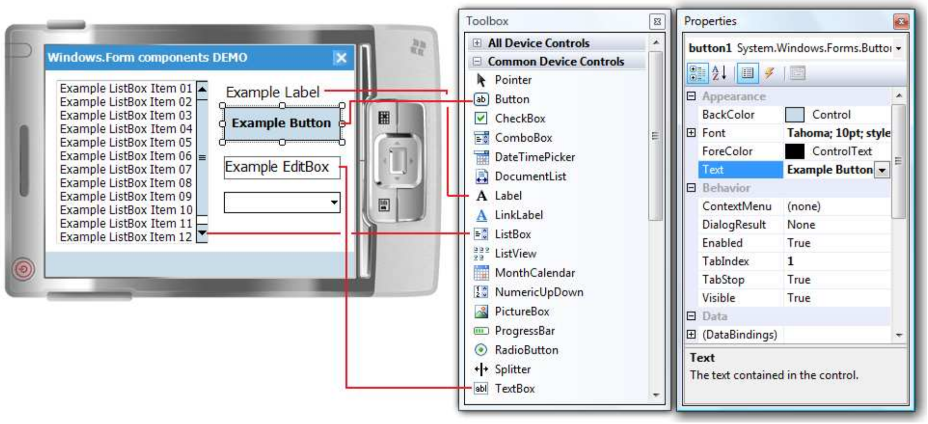
```
public class FormTabControlExample : Form
   {
    private System.Windows.Forms.TabControl tabControl;
    private System.Windows.Forms.TabPage tabPage1;
    private System.Windows.Forms.TabPage tabPage2;
    private System.Windows.Forms.Label label1;
    public FormHello()
       {
        //create the main tabControl and add it to the form.
        tabControl = new System.Windows.Forms.TabControl();
        tabControl.Location = new System.Drawing.Point(16, 16);
        tabControl.Size = new System.Drawing.Size(264, 240);
        Controls.Add(tabControl);
        //create the first tabPage and add it to the TabControl
        tabPage1 = new System.Windows.Forms.TabPage();
        tabPage1.Text = "tabPage1";
        tabPage1.Size = new System.Drawing.Size(256, 214);
        tabPage1.TabIndex = 0;
        tabControl.TabPages.Add(tabPage1);
        //create example Label on tabPage1
        label1 = new System.Windows.Forms.Label();
        label1.Location = new System.Drawing.Point(10, 10);
        label1.Text = "Example Label on tabPage1";
        label1.Size = new System.Drawing.Size(250, 100);
        tabPage1.Controls.Add(label1);
        //create the second tabPage and add it to the TabControl
        tabPage2 = new System.Windows.Forms.TabPage();
        tabPage2.Text = "tabPage2";
        tabPage2.Size = new System.Drawing.Size(256, 214);
        tabPage2.TabIndex = 1;
        tabControl.TabPages.Add(tabPage2);
        tabControl.SelectedIndex = 1;  //bringing tabPage2 to front
       }
   }
```

The `TabControl` contains tab pages, which are represented by `TabPage` objects, that we can add through the `TabPages` property. The order of tab pages in this collection reflects the order in which the tabs appear in the control. The user can change the current `TabPage` by clicking one of the tabs in the control. We can use

this control to load the form with many controls while allowing the user to switch quickly between different tabs. The currently displayed `TabPage` can be also changed programmatically by using one of the following `TabControl` properties: `SelectedIndex` or `SelectedTab`. The code example in listing 3-3 illustrates creation of a `TabControl` with two tab pages. Each tab page can contain several controls. In this example, only one `Label` is placed on `tabPage1`. By default, `tabPage1` (with index property equal to 0) will be selected. The last line of the code, use the `SelectedIndex` property to force bringing `tabPage2` to forefront.
Other useful controls are:

- `Label` – a control used to display textual information or description of other controls. `Text` property enables setting and retrieval of displayed textual content .
- `Button` – a class representing a Windows button control. Basic properties are: `Text`, `Enabled`, `Visible` and `BackColor`. The most useful events are the `Click`, `GotFocus` / `LostFocus`, `KeyDown`. Illustration of how to set up a new event handler for the button can be found in listing 3-2 (in lines ⑦⑧).
- `TextBox` – a control that enables the user to enter a string into a text field. `PasswordChar` property can be set to mask all entered chars. The most interesting event is the `TextChanged`, which is usually used to filter out or enable/disable another controls related to the entered content.
- `ListBox` – represents a Windows control for displaying a list of items. It is one of the most frequently used controls on mobile devices, because it enables to avoid onerous data readings (from key-pad or SIP) by simple selecting of predefined or memorized data.

The Windows Forms designer can be used to build GUI applications using Windows Forms. It includes a palette of UI widgets and controls (including buttons, progress bars, labels, layout containers and other controls) that can be dragged and dropped on a form surface (Figure 3-5).



button1 System.Windows.Forms.Button

Appearance	
BackColor	Control
Font	Tahoma; 10pt; style
ForeColor	ControlText
Text	Example Button
Behavior	
ContextMenu	(none)
DialogResult	None
Enabled	True
TabIndex	1
TabStop	True
Visible	True
Data	
(DataBindings)	

Text
The text contained in the control.

Figure 3-5 Drag&Drop composition of User Interface in MS Visual Studio Designer

Exercise WM.ex3
Sending and Intercepting SMS Messages

Create a Windows Mobile application that allows to use an SMS as a messaging transport medium between two (or more) mobile devices, with minimal input from the user.

a) Automate the process of SMS auto-reply, informing that somebody will be unable to manage the message for some reason (e.g. illness or vacation)
b) Automate mass SMS correspondence – e.g. create an application informing the group of receivers about some event, by automatically sending SMS messages to phone subscribers selected from application database, based on some selection rule.
c) A client application, for a mass entertainment-event booking system, assisting the user in making a ticket reservation by SMS.

Windows Mobile operating system contains a number of application programming interfaces (APIs) that are exclusive to Windows Mobile - powered devices. One of the most widely used is the `Microsoft.WindowsMobile.PocketOutlook` library, which is responsible for accessing and managing personal information data (contacts, calendar, tasks, etc.) and other messaging functionalities typical for phone devices. In particular, it contains Windows CE Mail API (CEMAPI), which handles all e-mail and Short Message Service (SMS) functionalities. This library gives us the ability to send and intercept SMS messages and perform some basic automation of the messaging application. Pay attention that Windows Mobile 6 “Standard” or "Professional" Development Kit have to be installed to utilize SMS functionality (it is unsupported under WM “Classic” SDK, targeting devices that have a touch-sensitive screen but no phone capabilities, such devices were formerly known as Pocket PCs).

The main class for SMS services is `SmsMessage` which is derived from the base `Message` class. Below are listed 8 properties (in majority inherited from `Message`) which we will use to prepare a message content (see table 3-4):

Table 3-4 Basic properties of `SmsMessage` class

Property name	**Description of `SmsMessage` properties**
`Body`	Gets the SMS message’s text body.
`From`	Gets the `Recipient` who sent the message.
`ItemId`	Gets the the message’s Item ID.
`LastModified`	Gets the date the message was sent.
`Read`	Gets or sets the read state of the message.
`Received`	Gets the date that the message was received.
`RequestDeliveryReport`	Gets and set an indication whether a delivery report is requested.
`To`	Gets the collection of `Recipient` for the SMS.

Body property of SmsMessage can be a string of unrestricted length, but it should be taken into account, that an individual billed SMS message is 160 characters long. This limitation is usually worked around by splitting a long message across multiple SMS messages, which are then reassembled on the receiving device. But longer messages will obviously create bigger telephone charges. After message preparation, the SmsMessage method send() is used to immediately dispatch of the SMS (Listing. 3-4.②)

Listing 3-4 The operations to send a single SMS message

```
using Microsoft.WindowsMobile.PocketOutlook;

void SendSimpleSMS ()
    {
    SmsMessage msg = new SmsMessage("555-666-777","Message Text");    ①
    msg.Send();  // Send the SMS message                               ②
    }

void SendCompoundSMS()
    {
    SmsMessage s = new SmsMessage();
    Recipient r = new Recipient("John Smith", "333-444-555");
    s.To.Add(r); // Adding SMS recipents
    s.To.Add( new Recipient("Anna Nowak", "666777888") );             ③
    s.Body = "This is an example of automatically generated SMS";
    s.RequestDeliveryReport = true;                                    ④
    s.Send();
    }
```

In simple case, the SmsMessage constructor ① is sufficient to set all necessary properties. In more complex situation, we can send an SMS message to multiple recipients by passing several Recipient objects to the To collection③ or optionally we can request a delivery report for the message ④ when the recipient device acknowledges receipt of the message (the report will be received in the form of separate SMS in device Inbox).

When we test the code utilizing SMS messages on the device emulator, we can send a loopback SMS by using the fake phone number 14250010001. This allows us to avoid paying real network charges.

To intercept incoming SMS we will use objects of class MessageInterception that can intercept messages matching a rule defined in MessageCondition. We can set up a comparison rule to match messages beginning or ending with a specific phrase or messages containing the phrase in any location. When a matching message is received, you can then access the message properties, such as sender and body text, from your code.

```
// default comparison type is Equal, and the match is case sensitive
MessageCondition( MessageProperty property, string comparisonValue);

MessageCondition( MessageProperty property,
                  MessagePropertyComparisonType comparisonType,
                  string comparisonValue,
                  bool caseSensitive );
```

A `MessageCondition` rule consists of: a `property`, a comparison type, a string comparison value and a setting whether the match is case sensitive. As an analyzed `property` can be used either the message `Body`, `Sender` or `Subject` from `MessageProperty` enumeration. A `comparisonType` specifies the ways that you can match message property values with a filter criteria. It has to be one of the values from `MessagePropertyComparisonType` enumeration (see tab 3-5).

Table 3-5 Possible values of `MessagePropertyComparisonType`

Enumeration name	**Description of the value**
`Contains`	Part of the property value contains the filter criteria.
`EndsWith`	The property value finishes with the filter criteria.
`Equal`	The property value is an exact match.
`NotEqual`	The property value does not match.
`StartsWith`	The property value begins with the filter criteria.

The default value of last `caseSensitive` argument (if omitted) is `true`. For example, we can create a filter criteria to catch all SMS containing the word "mobile" (case sensitive) inside the message body:

```
new MessageCondition(MessageProperty.Body,
                     MessagePropertyComparisonType.Contains, "mobile");
```

Finally we have to use `MessageInterceptor` rule object ① (see listing 3-5) to enable SMS application launcher. Add `MessageCondition` ④ to the interceptor rule and register it ⑤, so that even if our application is not currently running on the device, the system will launch it, allowing to process the message by adequate handler ②.

There are two versions of `MessageInterceptor` constructor. First constructor is used to load an existing interceptor rules ③ and the second version is used to create and set up a new interceptor ⑤. `IsApplication-LauncherEnabled` method ③ informs which one of them should be used. There are also two variants of interception actions specifying how a Pocket Outlook manages an intercepted message. In case of `InterceptionAction.Notify`, Pocket Outlook notifies our interception application, creates a copy of that message for it to process, then immediately notifies the next interception application. In case of `NotifyAndDelete`, it notifies our application and deletes the original message when it has finished processing the message. In this second case, no more applications intercept this message. The following code example (listing 3-5) demonstrates the whole registering process and a simple event handler, which just displays notification `MessageBox` that interception of rule-matching SMS was detected.

```
Listing 3-5  Example of SMS message interception code
using Microsoft.WindowsMobile.PocketOutlook.MessageInterception;
MessageInterceptor rule;                                              ①
// example interception handler
void SMS_Received(object sender, MessageInterceptorEventArgs e)       ②
{
 MessageBox.Show(((SmsMessage)e.Message).Body, "SMS Received");
}
void RegisterSMSInterceptor(void)
{
if(MessageInterceptor.IsApplicationLauncherEnabled("Intercept_1"))    ③
    { //Load existing settings.
    rule = new MessageInterceptor("Intercept_1");                     ④
    }
    else
    { //Set up and register a new rule                                ⑤
    rule=new MessageInterceptor(InterceptionAction.NotifyAndDelete);
    rule.MessageCondition=new MessageCondition(MessageProperty.Body,
                              MessagePropertyComparisonType.Contains,
                              "mobile");
    //Enable current application to be launched (if not running)
    //when a MessageReceived event is raised
    rule.EnableApplicationLauncher("Intercept_1");                    ⑤
    }

    // activate SMS interception handler
    rule.MessageReceived +=                                           ⑥
                    new MessageInterceptorEventHandler(SMS_Received);
}
```

To enable automatic start of SMS-processing application, we have to call `EnableApplicationLauncher` which stores the rule in the registry for future use, and the application path. The last statement of the above code sets up an event handler that is called each time a matching message is received. The method `SMS_Received` receives a `MessageInterceptorEventArgs` object that contains details of the message. Because the message is of the type `Message`, we must cast it to the `SmsMessage` type to access all of the message properties.

Exercise WM.ex4
Data Persisting with a Mobile Database Server

Use the visual designer tools in Microsoft Visual Studio 2005 to define project data sources and bind them to controls in your graphical user interface (GUI).

a) Program the SqlCeResultSet object, which allows fast, updatable access to example datatable in a SQL Server 2005 Compact Edition database.

b) Implement a small database composed of three datatables (books, friends, lendings) keeping the information about books borrowed from mobile device owner's personal library.

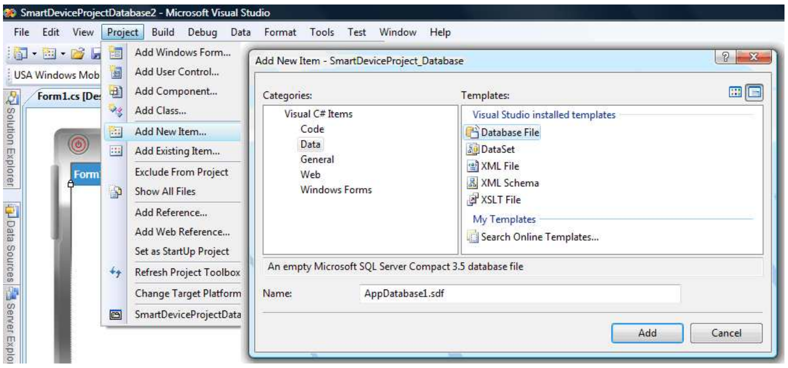
Most of mobile applications require some data to be permanently stored. The easiest way is to persist data in a simple binary file. But usually, more complex applications have to organize data in tables, provide some searching functionalities (using indexes to speed up the searching process), and representing relationships between data in different tables through foreign keys. This exercise is a short introduction to how Windows Mobile application can organize and persist data using Microsoft SQL Server Compact Edition (CE) database.

SQL Server Compact Edition is a lightweight relational database that supports data types that are compatible with full Microsoft SQL Server. It runs in-process in mobile application, so it does not require a separate server application to operate. At the beginning, this database solution branch was supported only on mobile devices and Tablet PCs. In result, it was called SQL Server Mobile Edition. Since the year 2005, it has been supported on all Windows desktop platforms like Windows 2000, Windows XP, Windows Vista, etc (including mobile devices). To emphasize the fact that it is a small database solution, which can be used not only on mobile devices, it has been renamed to Compact Edition.

To run an application that uses SQL Server CE, the suitable runtime software has to be installed on the mobile device. All Windows Mobile 6 and newer devices come with SQL Server CE already installed, but in the case of earlier OS versions, we have to ensure that the server components are downloaded before launching the application. The SQL Server CE runtime comes in three .cab files, which can be found in SmartDevices\SDK\SQL Server\Mobile\v3.0 subdirectory of Microsoft Visual Studio installation base. The installation can be done by copying the files onto the mobile device memory and opening them in File Explorer.

In the case of database application debugging performed on mobile device emulator, these additional operations are not necessary. Visual Studio automatically installs the run-time components on emulator (or development device) when we debug an application that uses SQL Server CE.

The firs step of the exercise, is to create an example instance of mobile database containing two tables (`Products` and `ProductCategories`) and add it to our application project. This can be done by using the Add New Item dialog (from Project menu), with selected Category: Data, and Template: Database File (see figure 3-6).



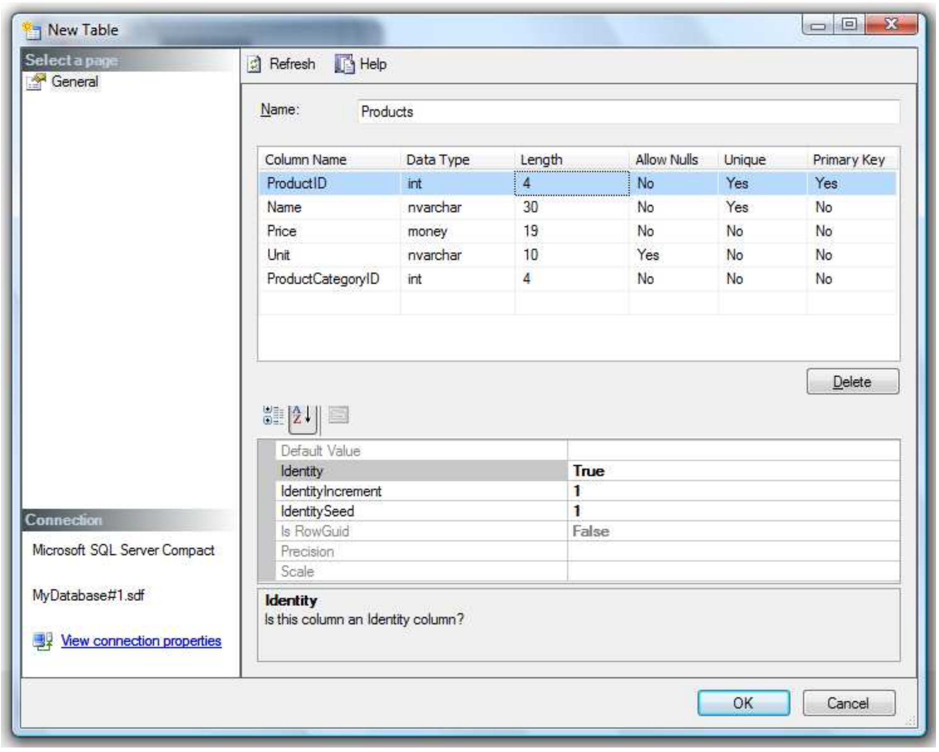
An empty Microsoft SQL Server Compact 3.5 database file

Name: AppDatabase1.sdf

Figure 3-6 Creation of new compact database with Add New Item option.

The second way is to click Connect to database icon from Server Explorer window, or click Connect To Database on the Tools menu. In this second case, we can alternatively connect to existing database (using button Browse) or connect to a newly created database (using button Create). During database creation we can specify a database password, and additionally we can specify that the data in the database is encrypted. To simplify the laboratory exercises, we can omit both these options, but it is strongly recommended to use them in all commercial applications.

As a creation result, in a Server Explorer window we can see new database instance under Data Connections category with the name which was given in the first step (MyDatabase#1.sdf in our case). Now we create two tables: one called `ProductCategories` that stores details of different categories in example product catalog and one called `Products` that contains details of individual products. In the Server Explorer window we expand the folders under the connection to our database, right-click the Tables folder, and click Create Table option. In the New Table dialog box, we can define the columns suitable for our `Products` table. It could be: `ProductID`, `Name`, `Price`, `Quantity` and `ProductCategoryID` to create relation with `ProductCategories` table. All the columns have to be initialized with suitable database parameters (for example: ProductID, int, 4, not null, unique, primary key, identity) as we can see on Figure 3-7.



Name: Products

Column Name	Data Type	Length	Allow Nulls	Unique	Primary Key
ProductID	int	4	No	Yes	Yes
Name	nvarchar	30	No	Yes	No
Price	money	19	No	No	No
Unit	nvarchar	10	Yes	No	No
ProductCategoryID	int	4	No	No	No

Default Value	
Identity	True
IdentityIncrement	1
IdentitySeed	1
Is RowGuid	False
Precision	
Scale	

Identity
Is this column an Identity column?

Figure 3-7 "New Table" dialog box used to set up columns definition for Products database

In the case of the `ProductCategories` table, we create only two columns: `ProductCategoryID` (int, 4, not null, unique, primary key, identity) and `Name` (nvarchar, 40, not null). After successful creation of both tables we can connect them with relation based on `ProductCategoryID` primary key. Visual Studio IDE is not a dedicated database design environment, so it does not offer any direct tool to create new database relations. It can be done indirectly, by executing following SQL query against the database:

```
    ALTER TABLE Products
    ADD CONSTRAINT Product_ProductCategory_FK
    FOREIGN KEY (ProductCategoryID) REFERENCES
    ProductCategories(ProductCategoryID)
    ON DELETE CASCADE
    ON UPDATE CASCADE
```

To execute the above (or any other) SQL query, right-click the name of processed database in Server Explorer, and then click New Query, which opens the Query Graphical Designer. The Designer is typically used to build SELECT statements to read data from the database. At this time, close the initial Add Table dialog box, and then copy the above SQL query in the place of automatically generated SELECT FROM statement in the query pane. Finally, right-click the query pane, and then select Execute SQL command from pop-up menu.

At this stage, both created tables possess automatically generated index files to assist database lookups connected with primary keys. Other index files, can be created by right-clicking Indexes category under the selected table, and executing Create Index option from the pop-up menu. In the New Index dialog box, we give the index a suitable name, and click Add to select the column that create the index. In this way, we can create an index on the foreign key field in a child table to help searching of all `Products` for selected `ProductCategory`.

After completing all operations described above, we have an example mobile database instance ready to process every standard database operation. Listing 3-6 illustrates example code which enable to execute a SQL query against database which is placed in the same place as the binaries of the application and is protected with example "pass" password.

Listing 3-6 Example code executing SQL querry against SQL CE database

```
using System.Data.SqlServerCe;
using System.IO;

public static void ExecuteQuery( String commandText )
{
   // Set up the connection string
   string databasePath = Path.GetDirectoryName(
            Assembly.GetExecutingAssembly().GetName().CodeBase );

   string connString = "Data Source=" + databasePath +
                       "\\MyDatabase#1.sdf; Password=pass";
```

```
    using (SqlCeConnection conn = new SqlCeConnection(connString) )
    {
       using (SqlCeCommand cmd = new SqlCeCommand(commandText, conn))
       {
           conn.Open();
           cmd.ExecuteNonQuery();
           conn.Close();
       }
    }
}
```

To facilitate code development for interactions between database and application user interface, we can use the visual designer tools from Microsoft Visual Studio. Visual designer automates definition of project data sources and binding them to controls in the graphical user interface (GUI).

The easiest way to work with data sources is through the Data Sources dialog box. We can open this dialog box by clicking Show Data Sources or Add Data Source link on the Data menu. In our case, a data source has not been yet defined, so we should use the second option.

With the Data Source dialog, we have to choose between: database, Web service, or object as the source of our data. We should choose Database and then bind to our example MyDatabase#1.sdf CE database. The Data Source Configuration Wizard displays the Choose Your Database Objects page on which we select the tables and views to include in the data source. By default, the wizard builds a strongly typed `DataSet` object and adds it to the project in the form of an XML schema (.xsd file). Instead of `DataSet`, we can also use the strongly typed `SqlCeResultSet` object (or both of them simultaneously) by changing Custom Tool property of _MyDatabase_1DataSet2.xsd file (in Solution Explorer window) from initial MSDataSetGenerator to MSResultSetGenerator or MSDataSetResultSetGenerator.

Finally, we can switch to Data Sources window, assign DataGrid visualization type to `Products` table, and drag-n-drop the `Products` onto `Form1` surface of the application user interface. The wizard adds `ProductBindingSource` to `Form1` resources and automatically generates `Products` data grid, which enables to browse the data contained in `Products` table (see Figure 3-8).

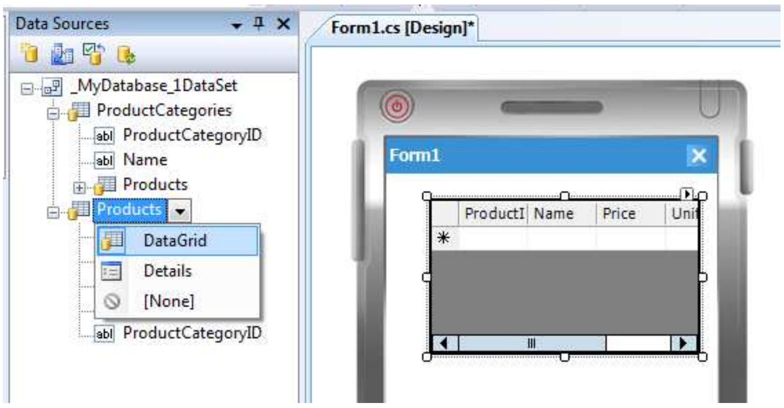


Figure 3-8 Automatic generation of DataGrid user interface assigned to Products DataSet

Clicking the SmartTag menu ▸ in the right-upper corner of the DataGrid component (in the Form1 designer window) enables us to select some design options, one of which is Generate Data Forms. By executing this option, the Visual Studio Forms Designer generates a set of forms to manipulate the records from a DataTable: to add new records, and to view or edit existing ones. This tool is very useful for building quick test programs, which is very comfortable during time-restricted laboratory exercises. More complete examples and discussions of Compact Edition database server can be found in "Chapter 3: Using SQL Server 2005 Compact Edition and Other Data Stores" of the "Mobile Development Handbook" [7]. In this exercise we have only described the solution where application data is stored directly on the device. In real enterprise world, most applications are not isolated and running entirely self-contained on a mobile device, with no need to communicate with the outside world. Usually, they are mobile components of a bigger enterprise solution. Chapter 7 of the abovementioned handbook, discusses how to synchronize data between applications running on the Windows Mobile device and backend servers, or fetch and store data in a SQL Server database on a network.

Exercise WM.ex5
Using the GPS to Track Geographic Position of Device

Develop a location aware application that obtains mobile device context (current geographic position) through GPS lookup and demonstrates some context-aware behavior (i.e. use the context passively or continually adapt/reconfigure itself to the new contextual information, to better solve the problem at hand).
Example context-aware behavior could be:

a) Proximate selection – interaction technique where a list of objects (e.g. services, places, printers, options, commands, etc.) is presented in the way emphasizing items relevant to the user's context, making them easier to choose.
b) Context-triggered actions – application that is continually monitoring the context changes and automatically executes some services when the right combination of context is detected.

The availability of low-cost, and usually integrated, GPS hardware for Windows Mobile devices makes it possible to develop applications that are aware of their current physical location. Adding location awareness to your application opens a lot of possibilities for many different kinds of consumer applications. In this exercise, we will get familiar with programming techniques for retrieving user/device location by using GPS lookup. Then applications can use such geographic data for: mapping, location reporting or "geo-tagging" other items of data.

There are several ways to retrieve location information inside a Windows Mobile device. All consumer GPS receivers output data over a serial port in a text format devised by the National Marine Electronics Association (NMEA). The most ordinary approach is to establish a connection to the attached device by a serial port, or a virtual serial port for devices connected by Bluetooth, to receive incoming NMEA data, and then parse it yourself. Unfortunately, receiving and analyzing rough data from GPS is very laborious. To illustrate the complexity of such task, let us see `GpsPosition` structure (Listing 3-7).

Listing 3-7 The structure of GPS receiver output data in NMEA format

```
[StructLayout(LayoutKind.Sequential)]
public class GpsPosition
{
    internal int dwVersion;         // Current version of GPSID
    internal int dwSize;            // sizeof(_GPS_POSITION)
    internal int dwValidFields;     // Valid fields
    internal int dwFlags;
    internal SystemTime stUTCTime; // UTC according to GPS clock.
    internal double dblLatitude;    // Degrees latitude
    internal double dblLongitude;   // Degrees longitude
    internal float flSpeed;         // Speed in knots
    internal float flHeading;       // Degrees heading
    internal double dblMagneticVariation;  // Magnetic variation
    internal float flAltitudeWRTSeaLevel;  // Altitute to sea level
    internal float flAltitudeWRTEllipsoid; // Altitude to ellipsoid
    internal FixQuality fixQuality;        //Quality of this fix
    internal FixType fixType;              // Is this 2d or 3d fix
    internal FixSelection selectionType;   // Auto or manual select.
    internal float flPositionDilutionOfPrecision;  //Position Dilut.
    internal float flHorizontalDilutionOfPrecision;//Horizontal Dil.
    internal float flVerticalDilutionOfPrecision;  //Vertical Dil.
    internal int dwSatelliteCount;         // Number of satellites
    internal SatelliteArray rgdwSatellitesUsedPRNs;//PRN num.of sat.
    internal int dwSatellitesInView;       // Satellites in view
    //PRN numbers of satellites in view
    internal SatelliteArray rgdwSatellitesInViewPRNs;
    // Elevation of each satellite in view
    internal SatelliteArray rgdwSatellitesInViewElevation;
    // Azimuth of each satellite in view
    internal SatelliteArray rgdwSatellitesInViewAzimuth;
    // Signal to noise ratio of each satellite
    internal SatelliteArray rgdwSatellitesInViewSignalToNoiseRatio;
    . . .
}
```

To make the work easier, Microsoft provides the GPS Intermediate Driver (GPSID), which is a software layer that sits between applications and the device driver for GPS hardware. This layer of abstraction allows applications to be written once and work with multiple GPS devices. The GPS Intermediate Driver API is exposed through a native code library. You can gain access to this library from managed code by using the sample that is included with the Windows Mobile Professional SDK (in subdirectory …\Samples\PocketPC\CS\GPS). Important benefit from GPSID, is that multiple applications can use the same physical GPS hardware. It was not possible with devices earlier than Windows Mobile 5.0. In those earlier Windows Mobile devices, only one application at a time could use GPS hardware.
To make use of GPSID, you have to add a reference to the abovementioned output assembly to your project and add the following using directive to your code.

```
using Microsoft.WindowsMobile.Samples.Location;
```

Once you have added a reference to the sample assembly to your application, you can access the GPS device through the `Gps` object. The `GpsDeviceState` and `GpsPosition` helper objects store information about the GPS device and your current location.

```
   GpsDeviceState device = null;
   GpsPosition position = null;
   Gps gps = new Gps();
```

The `gps.Opened` property and methods `gps.Open()` / `gps.Close()` should be used to activate and deactivate GPS receiver:

```
   if (!gps.Opened)
   {
       gps.Open();  // activates GPS receiver
   }
 . . .
   if (gps.Opened)
   {
       gps.Close();  // deactivates GPS receiver
   }
```

The location information can be retrieved synchronously, by direct call of method `GetPosition, e.g.:`

```
position = gps.GetPosition();
```

which returns an object of class `GpsPosition` with public properties:

```
float  Heading;        // degrees heading, North=0
double Latitude;       // latitude in decimal degrees, North positive
double Longitude;      // longitude in decimal degrees, East positive
bool   HeadingValid;
bool   LatitudeValid;  // validity of the above properties
bool   LongitudeValid;
```

Be careful, this method blocks until a position reading is returned from GPSID and with that from the underlying hardware. It waits until location data is received from one or more satellites. Second implementation of the method `GetPosition(TimeSpan maxAge)` gets the position reported by the GPS receiver, that is no older than the max age. If there is no data within the required age, `null` is returned. If `maxAge` is equal `TimeSpan.Zero`, then the age of the data is ignored. With it, you can retrieve locations more efficiently by returning a cached location reading as long as it is not older then the specified time. Following `UpdateData` method illustrates how GPS device state and retrieved position can be displayed on the screen in a internal text of `System.Windows.Forms.Label status` component.

Listing 3-8 Example of GPS data managing code

```
void UpdateData(object sender, System.EventArgs args)
{
   if (gps.Opened)
   {
      string str = "";
      if (device != null)
      {
         str = device.FriendlyName + " " +
               device.ServiceState + ", " +
               device.DeviceState + "\n";
      }

      if (position != null)
      {
          if (position.LatitudeValid)
          {
             str += "Latitude (DD):\n    " +
                    position.Latitude + "\n";
             str += "Latitude (D,M,S):\n   " +
                    position.LatitudeInDegreesMinutesSeconds +
                    "\n";
          }

          if (position.LongitudeValid)
          {
             str += "Longitude (DD):\n   " +
                    position.Longitude + "\n";
             str += "Longitude (D,M,S):\n   " +
                  position.LongitudeInDegreesMinutesSeconds +
                  "\n";
          }

          if (position.SatellitesInSolutionValid &&
              position.SatellitesInViewValid &&
              position.SatelliteCountValid)
          {
              str += "Satellite Count:\n   " +
              position.GetSatellitesInSolution().Length + "/" +
              position.GetSatellitesInView().Length +
              " (" + position.SatelliteCount + ")\n";
          }

          if (position.TimeValid)
          {
              str += "Time:\n   " +
                     position.Time.ToString() + "\n";
          }
      }
      status.Text = str;  // update the User Interface
   }
}
```

More interesting is the second position retrieving approach, by subscription to the `LocationChanged` and `DeviceStateChanged` events handlers, which generate continuing asynchronous updates when there is any change in the data that the GPS radio receives. The subscription can be set by commands:

```
gps.DeviceStateChanged +=
    new DeviceStateChangedEventHandler(gps_DeviceStateChanged);
gps.LocationChanged +=
    new LocationChangedEventHandler(gps_LocationChanged);
```

where `gps_LocationChanged` and `gps_DeviceStateChanged` are appropriate event handlers. Asynchronous updates are received on a separate thread.

Because it is not allowed to update User Interface controls on other threads than their creator, updating User Interface controls requires indirect `Invoke(updateDataHandler)` call to the presented `UpdateData` method.

```
private EventHandler updateDataHandler;
updateDataHandler = new EventHandler(UpdateData);

void gps_LocationChanged(object sender,
                         LocationChangedEventArgs args)
{
   position = args.Position;
   Invoke(updateDataHandler);  // update the User Interface
}

void gps_DeviceStateChanged(object sender,
                            DeviceStateChangedEventArgs args)
{
   device = args.DeviceState;
   Invoke(updateDataHandler);   // update the User Interface
}
```

In order to experiment with GPS Location retrieval, the mobile device should have either built-in or external GPS hardware or a running instance of FakeGPS application, installation cabinet file of which can be found in subdirectory Windows Mobile SDK\Tools\Gps.

4. Programming Symbian S60

The last series of exercises is dedicated to Symbian Operating System. On the contrary to the Java ME and Windows Mobile, which were evolutionally adopted from server and desktop solutions, Symbian OS has been developed for low power, small memory, mobile devices from the ground up. It has been derived from the "EPOC" operating system developed by "Psion Ltd." in the late 1980s and early 1990s, for Psion's "SIBO" (SIxteen Bit Organisers) devices. The main purpose of that company was to innovate and create a totally new branch of electronic devices called PDA (Personal Digital Assistant).

Psion-originated software was then adopted by Symbian Limited consortium, which was founded in 1998 by a group of major founders: Ericsson, Nokia, Psion, Motorola, and following shareholders: Matsushita, Siemens, Panasonic, Sony Ericsson, Samsung. The main goal of this organization is to develop and promote Symbian OS as a global industry standard for mobile phone systems and other WIDs (Wireless Information Devices). In result, Symbian has become a market-leading operating system and a de-facto industry standard for majority of wireless devices.

In 2008/9, the former Symbian Software Limited was acquired by Nokia and a new independent non-profit organisation called the Symbian Foundation was established to promote Symbian platform as a Open Source Software under the OSI- and FSF-approved Eclipse Public License (EPL).

Statistics published for the second quarter of 2010 showed that Symbian devices constituted a 41.2% share of smart mobile devices sold. There are estimates indicating that the cumulative number of mobile devices shipped with the Symbian OS, up to the end of Q2 2010, is about 390 million [12]. Nokia, in official documents and on the Web pages announces more than 250 million Symbian devices. Actually on the mobile market, we can observe a kind of "software war" in which the legacy smartphone platforms like Symbian have to keep up with mobile newcomers like iPhone or Android. In this grappling, the most undeniable advantage of Symbian is its huge penetration of the smartphone market and totally determined support from Nokia company.

4.1 Symbian OS versions and tools for C++ development

Symbian OS was designed to provide a consistent software platform for very broad range of phones. It scales from souped-up general use "feature phones" like 6120, through business-oriented phones like E71, to rich multimedia phones like N96. Before we proceed to Symbian software development, it is useful to understand which SDKs are available for particular devices. Nokia's phone devices can be split into several series named by following numbers from the sequence: Series 20, 30,40, 50, 60, 70, 80, 90. Currently the most widespread series, which use the Symbian as an operating system, is the series S60. In this course our focus will be on Symbian C++ programming for the S60 platform. Symbian devices can also be programmed using other programming languages/frameworks like: Python, Java ME, Flash Lite, Ruby, Visual Basic and Web Runtime (WRT) Widgets. But only native Symbian C++, which is a domain specific dialect of general purpose C++ language, give us the full access to the device capabilities. There have been four major releases of S60 SDK:

- S60 1st Edition (since year 2001) with Symbian OS v6.*, which has been supporting only SMS, EMS, MMS, J2ME, XHTML services, with strong restriction that the devices' display resolution has been fixed to 176x208 pixels.
- S60 2nd Edition (since year 2003) with Symbian OS v7.*, supporting scalable UI with multiple resolutions and orientations, which includes HTML browser extensions, support for EDGE and WCDMA (UMTS).
- S60 3rd Edition (since 2005) with Symbian OS v9.*, featuring several security improvements (Secure Platform) where a user may install only programs that have a certificate from a registered developer.
- S60 5th Edition" (since 2008) with Symbian OS v9.4, where the major feature is support for high-resolution 640x360 touch screens.

In the lab, we will utilize S60 SDK starting from 3th Edition and Symbian OS v9.*. There are three main integrated development environments available to Symbian OS C++ developers, namely:

- CodeWarrior, formerly the IDE of choice for Palm programmers and for PlayStation development, which is now being deprecated.
- Carbide.c++, an Eclipse-based IDE developed by Nokia. Offered in four different versions: Express, Developer, Professional, and OEM, with increasing levels of capability. The Express version is available for free and provides the necessary functionality to build and compile applications. The Developer version has additional tools for on-device debugging and graphical tools for Rapid Application Development
- Microsoft Visual Studio also supports Symbian development through the Carbide.vs plugin.

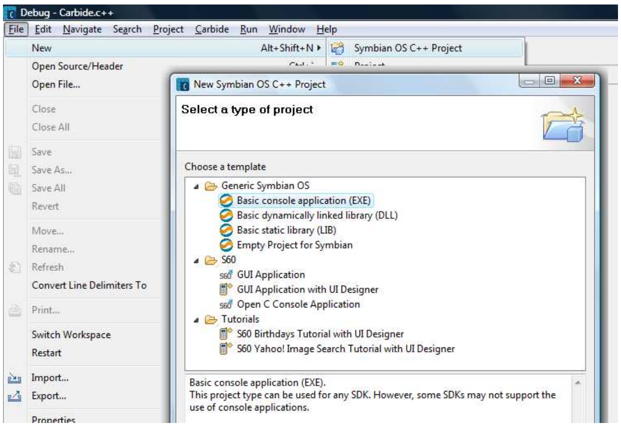
For early versions of Symbian OS, the commercial IDE CodeWarrior for Symbian OS was favoured. The CodeWarrior tools were replaced in 2006 by Nokia's Carbide.c++ which currently is the most recommended IDE. Visual Studio .NET is used by some groups of keen Microsoft Windows programmers. There is also a version of a Borland IDE for Symbian OS, but it is not very popular. The totally new Qt-Creator IDE emerged in 2009 and is receiving Nokia's very strong support. But this tool is more related to Trolltech Qt multiplatform library, rather than native Symbian C++ and will be separately presented in introduction to the last Qt-Symbian exercise of this course.

Correct configuration of Carbide.c++ and Series 60 SDK development environment requires some steps, which unfortunately will not be performed automatically. Firstly, the Carbide.c++ tools require an installation of Perl to run build scripts. ActivePerl installer can be downloaded from (http://www.activestate.com/). Be careful, ActivePerl version 5.6.1 is required by Symbian tools, and later versions will not work. Additionally, take into account, that you have to install Carbide.c++ to the same drive as Perl. Make sure that there are no space characters contained in any directory name utilized by Perl, Carbide.c++ or Symbian project workspace directory. If you allow spaces in directory names, your code can't be compiled. Secondly, from the page (http://www.forum.nokia.com/main/resources/tools_and_sdks/) download and install Carbide.c++, an IDE which is designed from the ground up for developers creating Symbian C++ and Qt software for Symbian devices. Choose Developer version 2.* or later. Finally, depending on the type of accessible target device, use the page (http://www.forum.nokia.com/main/platforms/s60/) to download the proper version of S60 SDK. In the lab, to speed up exercise pace, we will generally use device emulators, so it does not matter which S60 SDK will be chosen (3rd Edition FP1, 3rd Edition FP2 or 5th Edition). It is suggested, to use S60 3rd SDK FP2 for button-based application and S60 5th SDK 1.0 for touch-based applications.

4.2 Programming Exercises

Following the good practice of most programming tutorials, we will not start from programs with attractive graphical user interface, but utilize an old fashioned text-mode console applications. It is especially important in the case of Symbian OS, which is infamous for a steep learning curve. Symbian C++ programming requires the use of special techniques such as: leaves, two phase constructors, cleanup stack, descriptors and active objects. This can make even relatively simple programs harder to implement. According to some experiments performed by VisionMobile, as a part of "Developer Economics 2010" research, the Symbian platform takes on average 15 months or more to learn (two to three times more than other mobile frameworks) [12].

The goal of the first exercise is to get to know the basic data types of Symbian OS as well as the console mode of the Symbian OS emulator. Note, that there is no built-in way to execute a console application on a real Symbian device. To run a console application on the phone you have to launch it using either a program like FExplorer or a remote console like QConsole. Fortunately, Carbide.c++ environment provides special "Basic console application" wizard (see Figure.4-1) which will simplify the creation of template code for console application.


Debug - Carbide.c++
File Edit Navigate Search Project Carbide Run Window Help
New Alt+Shift+N
Open Source/Header
Open File...
Close
Close All
Save
Save As...
Save All
Revert
Move...
Rename...
Refresh
Convert Line Delimiters To
Print...
Switch Workspace
Restart
Import...
Export...
Properties
Symbian OS C++ Project
New Symbian OS C++ Project
Select a type of project
Choose a template
Generic Symbian OS
Basic console application (EXE)
Basic dynamically linked library (DLL)
Basic static library (LIB)
Empty Project for Symbian
S60
GUI Application
GUI Application with UI Designer
Open C Console Application
Tutorials
S60 Birthdays Tutorial with UI Designer
S60 Yahoo! Image Search Tutorial with UI Designer

Basic console application (EXE).
This project type can be used for any SDK. However, some SDKs may not support the use of console applications.

Figure 4-1 Setting up template code of text-console application in Carbide.C++ IDE

The structure of IDE generated template `*.cpp` file for console application is illustrated in Listing 4-1. The entry point of the application is the `E32Main()` function ⑦, which firstly builds the rest of the necessary Symbian framework (cleanup stack ⑧ and text console instance ⑨, pointer of which is stored in the global variable named `console` ③), than runs the main user created function `MainL()` ④, and finally will catch and output any errors (leaves) that the code might produce. Because user function `MainL()` might generate some leaves, its name is marked with suffix "L" and is called within TRAP harness ⑩.

Listing 4-1 Template source code of simple console application

```
// Include Files
#include <e32base.h>
#include <e32std.h>
#include <e32cons.h>   // Console class definition                    ①

// Text constants definitions
_LIT(KTextConsoleTitle, "Console window title");                      ②
_LIT(KTextFailed, " application failed, leave code = %d");

// Global Variables, in this case only console object
LOCAL_D CConsoleBase* console;                                         ③

LOCAL_C void MainL()                                                   ④
   { // add your program code here, example code below
     . . .
     console->Write(_L("Hello, world!\n"));                            ⑤

     _LIT(KTxtHello, "Hello, world! (once more)\n ");                  ⑥
     console->Printf(KTxtHello);
     . . .
   }

// Starting point of Symbian console application
GLDEF_C TInt E32Main()    // Global Function: E32Main                  ⑦
   {
    // Create auxiliary cleanup stack and output console object
    CTrapCleanup* cleanup = CTrapCleanup::New();                       ⑧
    TRAPD(createError,
          console = Console::NewL(KTextConsoleTitle,                   ⑨
                    TSize(KConsFullScreen, KConsFullScreen)));
    if (createError) { delete cleanup; return createError;  }

    // Run our code <MainL()> inside TRAP harness
    TRAPD( mainError, MainL() );                                       ⑩

    // Resolve eventual errors and clean up all auxiliary objects
    if (mainError)
       console->Printf(KTextFailed, mainError);
    delete console;
    delete cleanup;
    return KErrNone;
}
```

Other important elements of presented code are: inclusions of necessary C++ definitions (header) files ① and `_LIT()` macros converting text constants into a literal string descriptors ②⑥. Symbian implementation of `Printf()` requires a parameter as literal string (`TLitC` class object, which defines a number of operator overloads). In some cases the separately used `_LIT()` macro can be replaced by shorter inline `_L()` version which can be placed inside function call ⑤, but it works less efficiently, so the use of `_LIT()` is generally more recommended.

For the sake of code clarity, the unchanging parts of console template `main.cpp` source file (i.e. includes global console definition and `E32Main()` starting function) can be shifted to accompanying `main.h` C++ definition file. In such a case, the example code is more clear and much more similar to standard C++ main template (see Listing 4-2). We will use this approach in subsequent example code snippets and inside "fill-in" example projects utilized in the course of the laboratory classes.

Listing 4-2 Simplified version of Symbian console application with function E32Main() hidden in the file "main.h"

```
#include "main.h"

LOCAL_C void mainL(CConsoleBase* console) {
   . . .
   console->Write(_L("Hello, world!\n"));
   . . .
}
```

Advanced Symbian applications most likely make use of "active objects", which are another example of Symbian specific technique helping to ensure longer battery life (the CPU is powered down when applications are not directly dealing with events). To enable the use of active objects, an instance of `CActiveScheduler` has to be installed and activated before `MainL()` call. Usually it is done, by replacing the first call of `MainL()`, by the intermediate call of `DoStartL()` harness, which wraps up user `MainL()` function, with the code managing the `CActiveScheduler` creation and destruction (see Listing 4-3). But these operations can also be shifted to accompanying `main.h` definition file, so the final `main.cpp` source file will not change.

Listing 4-3 Installing CActiveScheduler to enable "active objects"

```
LOCAL_C void DoStartL() {
   // Create active scheduler (to run active objects)
   CActiveScheduler* scheduler = new (ELeave) CActiveScheduler();
   CleanupStack::PushL(scheduler);
   CActiveScheduler::Install(scheduler);

   MainL(console);

   // Delete active scheduler
   CleanupStack::PopAndDestroy(scheduler);
}
```

Exercise SYM.ex1
Symbian OS C++, Basic Classes, Naming Conventions

Create Symbian C++ console application which illustrates the use of Symbian specific data representation (numbers, chars, texts), demonstrates reading and writing of user data to text-console, and performs some elementary flow control instructions (conditionals and loops):

a) Define example constants and variables for every category of data type (integer and float numbers, single char, boolean and text data), set initial constant or store the results of calculated expressions.

b) Familiarize yourself with the help description of `CConsoleBase` class methods, which enable to read/write the data from console (`getch()`, `Printf()`, `Write()`) and control console cursor position (`SetPos()`, `WhereX()`, `WhereY()`, `ClearScreen()`). Write a text version of ping-pong simulation where flying ball is rebounding from display borders.

c) Use <e32math.h> library and `Math::Random()` function to generate random integer numbers. Engage the random generator to simulate variable disturbance of ping-pong motion simulation.

d) Test how to use objects of R-Type Symbian class. Define a local variable of the `RBuf` class, which owns string data on the heap and manages it automatically. `RBuf` is one of the Symbian OS descriptors which is the Symbian way of C++ strings representation. Use `CreateL()` and `Close()` methods to allocate and free the memory utilized for the `RBuf`. Remember! Do not use `new` or `delete` operators for R-Type variables.

Symbian C++ is a domain specific dialect of general purpose C++ programming language. It provides a number of Symbian specific standard type definitions that should be used instead of the native built-in C++ types (such as `int`, `float` or `char`). These definitions are guaranteed to be compiler-independent and should be included with `<e32def.h>`.

Symbian OS has its own naming convention for classes, variables and functions. Naming conventions are used to improve code readability, as they allow the programmer to determine the nature of an item without looking at its implementation. The first idea of using prefixes is similar to Hungarian notation, with different letters being used for different objects. There are six basic naming rules:

- the first letter of a class or function name should be capitalized,
- function parameters and automatic variable names should start with a lower-case letter,
- constants are prefixed with the upper-case letter 'K',
- non-static class member variables are prefixed with the lower-case letter 'i', referring to an `instance` of a class.
- function parameters are prefixed with the letter 'a' which stands for `argument`
- the name of an enumerated type is prefixed with 'T' (because it is a `Type`) and its members' names are prefixed with 'E' (from `Enumeration`).

Following listing (4-4) demonstrates how to use the above rules in example definitions of: a class, a global variable, a constant and an enumeration.

Listing 4-4 Illustration of Symbian C++ naming conventions

```
class TExampleClass   //prefix T → indicates a class/a new Type name
  {
   //upper case first letter → all Function names begin with upper
   void FunctionName( TInt aExampleArgument );
   //prefix a → indicates an argument of a function
   //prefix i → indicates an instance / member variable of the class
   TInt iExampleInstance;
  };
```

```
TExampleClass exampleObject;     //lower case letter → any variable

#define KExampleConstant 100     //prefix K → indicates constant
const TInt KMaxSize = 256;

enum TExampleEnumeration (EFirst, ESecond, EThird);   //Enumeration
```

Another naming convention is related to Symbian's classes definition. Every class has it's own prefix (i.e C, T, R or M), which denotes the nature of the class and makes the creation, usage and destruction of objects more straightforward. There are five class types:

- **T-class** – A simple class that does not use dynamic data (heap-allocated memory). It behaves much like the C++ built-in types and can be seen as an equivalent of a C++ `struct`. As no cleanup is required, there is no need for a destructor. All Symbian basic types defined in file `<e32def.h>` also have the prefix **T**. Below there are the tables describing the basic type characteristics:

  Table 4-1 Symbian basic types defined in file `<e32def.h>`

Signed integer	
Name	Bytes
TInt	≥4
TInt8	1
TInt16	2
TInt32	4
TInt64	8

Unsigned integer	
Name	Bytes
TUint	≥4
TUint8	1
TUint16	2
TUint32	4
TUint64	8

Void	
Name	Type
TAny	Void

Boolean	
Name	Size
TBool	4

Floating point	
Name	Size
TReal	8
TReal32	4
TReal64	8
TRealX	12

Character	
Name	Size
TText8	1
TText	2
TText16	2
TChar	4

  Note, that the use of floating-point numbers should be avoided because they are not natively supported and floating point calculations may seriously slow down application execution.
- **C-class** - A class that derives from `CBase` or another class already derived from `CBase`. The 'C' prefix indicates that the class is constructed on the `heap`. Unlike T-classes, C-classes own pointers to other objects, and have a destructor to clean up these member variables. Such classes should be used with caution to avoid memory leaks.
- **R-class** - The R prefix indicates that such class holds an external `resource` handle, for example a handle to a server session. It will typically have a constructor to set this resource handle to `null`. The object must be associated with a resource by calling initializing methods such as `Open()`, `Create()` or `Initialize()`. There are corresponding `Close()` or `Reset()` methods to free resources, otherwise memory leaks can occur. A common mistake when using R classes is to forget to call `Close()` or to assume that there is a destructor which cleans up the owned resource.
- **M-class** – "mixin" class - defines an abstract interface (class consisting of purely virtual functions). Usually used to define callback interfaces or observer classes. Important issue involving inheritance list is that the first implemented class should be `CBase` and after that will come the M-interfaces.

- **S-class** - An ordinary C++ struct (without any member functions) should be prefixed with a uppercase S.

The last set of naming rules uses single-letter suffixes in function names, to signalize possible changes in program flow or clarify heap memory state after function execution:

- The suffix '**L**' is used to indicate that the function may leave.
- The suffix '**C**' indicates that the function will return a pointer that has been pushed onto the cleanup stack.
- The suffix '**D**' indicates that the calling object will be deleted from heap by called function.

Exercise SYM.ex2 Symbian OS exception handling mechanisms (Leaves, Traps and Cleanup Stack)

Create your own example code illustrating typical application of Symbian exception handling techniques, to manage memory resource allocation fails and utilize CleanStack to protect your application against memory leaks,

a) Intentionally trigger a Leave inside different points of your application code (e.g. by a direct call to User::Leave(KErrGeneral) method) and debug changes in execution sequence caused by the Leave occurrence.

b) Create an example code which dynamically allocates several (at least two) CBase inherited objects, with a random possibility of a Leave generation. Utilize CleanupStack mechanism to protect allocated memory against orphaning. Check the state of the heap memory preservation during your code execution with macros: __UHEAP_MARK and __UHEAP_MARKEND.

Important aspect of Symbian OS programming is effective exceptions handling and smart allocation of very limited resources of a mobile device. Traditionally, C++ programming language employs conventional (but resource consuming) paradigm of try/throw/catch sequence. Unfortunately this mechanism is unsupported in Symbian environment (before version 9) since it was considered too much memory intensive at the time the Symbian OS was designed. Instead, Symbian C++ introduces lighter programming concepts: Leave, TRAP and CleanupStack, that will help us to maintain memory-leak free code. In general, working with leaves and traps is very similar to using throws. A Leave can be perceived as an equivalent to a throw, and a TRAP as an equivalent to the try and catch statements. A Leave can occur in three ways:

- By explicit call of the variant of system function User::Leave()

```
IMPORT_C static void  Leave(TInt aReason);
IMPORT_C static void  LeaveNoMemory();
IMPORT_C static TInt  LeaveIfError(TInt aReason);
IMPORT_C static TAny* LeaveIfNull(TAny* aPtr);
```

- by calling another function which can do `Leave` (the name of such function is traditionally marked with suffix L).
- by use of the Symbian C++ overloaded form of operator `new`, which takes `ELeave` as a parameter (which generates `Leave(KErrNoMemory)` if there is not enough memory to allocate the object)

At the point of the leave, the program execution stops and returns immediately to the `TRAP`, from within which the function was called. As additional information, an error code (integer value) is returned. Symbian provides two trap macros: `TRAP` and `TRAPD`.

```
TRAP( ErrorVariable, PotentiallyLeavingFunctionL() );
TRAPD( ErrorVariable, PotentiallyLeavingFunctionL() );
```

The difference between `TRAP` and `TRAPD` is that, in the first case we must define the `TInt ErrorVariable` object by ourselves while the second macro `TRAPD` (with suffix D, from Definition) will do it instead of us. An illustration of `TRAPD` use is shown below:

```
. . .
TRAPD( err,ExampleFunctionL() ); //trapped call of a leaving function
if( err != KErrNone )
   {
    //a Leave occurred → exception handling have to be done here
   }
. . .
```

TRAP macros can also be nested to catch and handle leaves at different levels. A typical reason of a leave is a resource allocation fail. In the case of mobile devices such fail is usually caused by memory allocation under low-memory conditions. Standard C++ solution is to check validity of returned pointer after every allocation call:

```
CTypeClass *pointer = new CTypeClass; //potential allocation fail
if( pointer==NULL )                   //check the pointer validity
   {
    User::Leave(KErrNoMemory);        //signalize an error by a Leave
   }                                  //with KErrNoMemory err code
```

To avoid tedious repetition of the above sequence, Symbian C++ provides an overloaded version of the `new` operator (recognized by a parameter `ELeave`) which will automatically generate a `Leave` with `KErrNoMemory`, if it fails to allocate the required memory. In result, an equivalent of the above allocation code will have the one line form:

```
CTypeClass *pointer = new(ELeave) CTypeClass;
```

From Symbian OS version 9.1 leaves are implemented in terms of standard C++ exceptions, but still leaves are the fundamental part of Symbian error handling and are used throughout the system.

CleanupStack

Symbian platform was designed to be used on devices with limited memory that are often not rebooted for weeks or months. In result, such applications are left running almost indefinitely, rather than being closed after every use. They are vulnerable to memory errors. If resources are not properly cleaned up in the event of a leave, then memory leaks will occur. Such memory loss will accumulate with time and finally the system will crash.

CleanupStack is a static resource (stack) that can be used for safe handling of pointers to dynamically allocated memory on the heap, which could be lost in the case of a Leave occurrence. The GUI applications and servers both have cleanup stacks created for them as part of their respective frameworks. Our simple console programs, created for test and demonstration purposes, require a cleanup stack to be explicitly allocated for every thread which uses cleanup processing. Listing (4-1⑧) illustrates typical CleanupStack framework for console application with E32Main root function. Note that the cleanup stack variable can have any name (in our case cleanup) because it is only used to reference the stack when it needs to be destroyed. All other calls to the active stack will be done through the static functions from class CleanupStack. In the CleanupStack there are three overloads of the PushL() method:

```
static IMPORT_C void PushL(TAny* aPtr);
static IMPORT_C void PushL(CBase* aPtr);
static IMPORT_C void PushL(TCleanupItem anItem);
```

where PushL() argument determines how the item is destroyed when it is cleaned up in case of a leave or a call is made to CleanupStack::PopAndDestroy(). In the case of anonymous TAny* pointer, the memory referenced by the pointer is simply deallocated by invoking User::Free(), delete is not called on it, so no destructor is invoked. In the case of CBase-derived object, cleanup operation is performed by invoking delete operator, thus calling the virtual destructor of the CBase-derived object. TCleanupItem can be used to allow cleanup to be more sophisticated than simply deleting objects, for example, releasing access to some shared resource.

If a Leave occurs while an object pointer is on the stack, it is cleaned up automatically. Otherwise it has to be manually removed by direct call of one of eight Pop function variants:

```
static IMPORT_C void Pop();
static IMPORT_C void Pop(TInt aCount);
static IMPORT_C void PopAndDestroy();
static IMPORT_C void PopAndDestroy(TInt aCount);
static inline void Pop(TAny* aExpectedItem);
static inline void Pop(TInt aCount,TAny* aLastExpectedItem);
static inline void PopAndDestroy(TAny* aExpectedItem);
static inline void PopAndDestroy(TInt aCount,TAny* aLastExpectedItem);
```

The general CleanupStack protecting procedure goes as follows:

1. Before calling a potentially Leaving function, we have to call CleanupStack::PushL(pointer) to add every locally scoped pointer to cleanup stack.
2. If the function does not leave, just after function return, we must place the code which removes added pointers from the stack, by calling CleanupStack::Pop(). Alternatively we can merge Pop stack

operation with simultaneous `delete` of pointed resources from the heap by call of `CleanupStack::PopAndDestroy()` method.

3. In the event of a `Leave`, program execution returns immediately to the `TRAP` (from within which the function was called) and all heap items, that were memorized on the `CleanupStack`, since the last `TRAP` are freed.

Symbian C++ provides a set of debug-only macros that can be added directly to code to check that memory is not leaked. The most commonly used are defined as follows:

```
#define   __UHEAP_MARK      User::__DbgMarkStart(RHeap::EUser)
#define   __UHEAP_MARKEND   User::__DbgMarkEnd(RHeap::EUser,0)
```

The macros verify that the default user heap is consistent. If heap cells have not been freed, a panic is raised and application stops. Calls to this macros can be nested, but every time each call to `__UHEAP_MARK` must be matched by corresponding call to `__UHEAP_MARKEND` (see Listing 4-5).

Listing 4-5 Utilizing __UHEAP macros to check heap preservation at the time of E32Main() execution

```
GLDEF_C TInt E32Main()
  {
   __UHEAP_MARK;  // marking initial state of the memory heap

   // any code which allocate memory on the heap,
   // in this case creation of CleanupStack and example function call
   CTrapCleanup* cleanup=CTrapCleanup::New();
   TRAPD(error,callExampleL());
   delete cleanup;
   // at this point application is going to be terminated,
   // all allocated memory should be released

   __UHEAP_MARKEND; // checking if the initial state was recovered
   return 0;
  }
```

Note. The function name suffixes are not checked during compilation. It is possible to forget to append suffix 'L' to a function that leaves, especially when modifying previously non-leaving functions. Such oversight does not generate immediate error, but critically affects the readability of the code. Symbian provides a tool called LeaveScan, that checks code for incorrectly named leaving functions. This tool should be used regularly on large projects created by a team of several developers.

Exercise SYM.ex3
User Interface and Event Handling with Qt library

a) Create your own Qt demo application, demonstrating possibilities of User Interface construction from basic Qt Widgets: `QLabel`, `QLineEdit` and `QPushButton`.

b) Create your implementation of general purpose number calculator. As a starting point, use `CalculatorWidget` definition from listing 4-9.

Classical Nokia's User Interface layer, on top of the Symbian OS, was called "S60". In fact, there are several different user interfaces available for Symbian OS, which are designed to allow manufacturers to produce a range of phones in different styles. They can be generally divided into three categories: S60 (formerly known as Series 60), Series 80, and UIQ. There was a fourth category, Series 90, but it has been now merged into S60.

S60 User Interface consists of a suite of libraries and is intended to be provided with fully-featured modern mobile phones with large color screens, which are usually referred to as smartphones. S60 is most often associated with Nokia, but it is also licensed to other mobile-phone manufacturers such as: BenQ–Siemens, Samsung and Panasonic.

Standard approach to create S60 UI is to utilize rather complicated native Symbian C++ framework architecture consisting of: an application class (inherited from `CEikApplication`), a document class (inherited from `CEikDocument`) and an application user interface class (inherited from `CAknAppUi`), several views or containers classes (inherited from `CCoeControl`). The set of these classes creates the fundamental application behaviour and is mixed with other Symbian specific programming techniques like: leaves, cleanup stack, two-phase constructors, etc. As a result, the final application structure and application code is hardly readable for most beginners in Symbian C++ programming. It has been slowly becoming evident, that techniques, developed for the much more restricted mobile hardware of the 1990s, simply caused unnecessary complexity in source code. Current Nokia's strategy, to overcome this problem is to utilize standard C++, with the totally new Qt SDK. In this and following exercises, we will get familiar with what the "Qt" is and how it can be used for efficient development of applications and user interfaces for S60.

Qt Framework

Qt - pronounced as cute (/kju:t/) or as Q.T. (/'kju:ti/), is a cross-platform application framework that is widely used for developing application software with Graphical User Interface. In fact, the Qt is not so new, because it has been available on the market since 1996. It has been developed and marketed by Qt Software, formerly known as Trolltech (the Norwegian company which is the original producer of Qt). Nokia acquired Trolltech in 2008 and renamed it to "Qt Development Frameworks".

Qt uses standard C++ but makes extensive use of a special code generator. It runs on all major platforms and has extensive internationalization support. Non-GUI features include: SQL database access, XML parsing, thread management, network support, and a unified cross-platform API for file handling. Qt is free and open source software. It is distributed under the terms of the free GNU General Public Licence and Lesser General Public Licence.

Qt is released by Nokia for the following platforms: Linux/X11, Mac OS X, Windows, Embedded Linux, Windows CE / Mobile, Symbian, Maemo and MeeGo. According to many opinions on different programmer's forums, the Qt greatly reduces development effort through intuitive APIs that "deliver more functionality from less code".

Qt programmers can use the integrated development environments they are most comfortable with because Qt has been integrated with several of the most

commonly used IDEs. These include Microsoft Visual Studio .NET, Eclipse, and Carbide.c++. Qt Software also provides Qt Creator, which is a lightweight IDE tailored specifically to the needs of Qt developers. During the lab we will utilize Nokia Qt SDK with included Qt Creator, which can be downloaded from page: http://www.forum.nokia.com/Develop/Qt/Tools/
Note, that the Linux distribution of Nokia Qt SDK does not have tools for Symbian development. For Symbian Qt programming Windows XP or Vista is needed.

The Qt framework has been designed with ease of development in mind. It is a cross-platform application framework which means that applications written with Qt can be used in many target platforms. The Qt API is implemented in C++ and expands this programming language with a meta-object model which adds following new features to C++:

- Qt class library which contains about 700 classes with hierarchical object-oriented architecture,
- object to object communication mechanism called "signals and slots",
- events, event filters and timers,
- string translation to support internationalization,
- guarded pointers that are automatically set to NULL when the referenced object is destroyed,
- class type identification and dynamic casting without using the C++ run-time type identification (RTTI),
- dynamic and discoverable object properties, which allow interaction with classes unknown during the compile time.

QApplication is the central class of every Qt application, that handles the event loop and the main settings of application. It is responsible for managing application events such as keyboard, mouse or touch events on touch screen devices. It also manages the screen size as well as the font and style used by the application.

QWidget is the basic user interface element in Qt and it is inherited from QObject. It receives mouse, keyboard and other events from the windowing system and does the widget painting or drawing on the screen. All user interface elements: buttons, labels, views inherit from QWidget. Listing 4-8 illustrates the general structure of "Hello, world" Qt demonstration for Symbian S60 and other platforms.

Listing 4-8 Qt "Hello, world!" demonstration code

```
#include <QtGui/QApplication>                                  ①
#include <QtGui/QLabel>

int main(int argc, char *argv[])                               ②
{  QApplication app(argc, argv);                               ③
   QLabel label("Hello, world!");                              ④
   #if defined(Q_WS_S60)
      label.showMaximized();                                   ⑤
   #else
      label.show();
   #endif
   return app.exec();                                          ⑥
}
```

As we can see, Qt looks like standard C++ code. A the beginning the required classes are included ①. Including the classes directly like `#include <QApplication>` also works, but the recommended convention is the use of the Qt module name in front. The entry point for Qt applications is function `int main()` like for normal C++ programs ②. In fact, every Symbian program enters execution from `TInt E32Main()`, but the Qt for S60 port handles this conversion automatically through a library called `libcrt0`. For GUI programs we need to create a `QApplication` object that initializes program settings and later will handle the application event loop ③. Then, we create one `QWidget`, or more precisely, a `QLabel` which is for showing one label with the text "Hello world!" ④. All the `QWidgets` can take another `QWidget` in the constructor to become its parent. This label does not have a parent, which means that it will become a window of its own. As Widgets are not visible by default, we make this label visible ⑤. In the case of S60 mobile device, the application window should occupy the whole screen, so for S60 the label will be maximized. For other platforms simple `label.show()` will be better solution. Finally, we enter the actual execution by calling `exec()` for the `QApplication` object. This starts the applications event loop. When the window is closed, the `exec()` function is exited and we exit the main function as well.

As a more advanced example of user defined Qt Widget we can use CalculatorWidget (see listing 4-9)

Listing 4-9 Example implementation of user defined CalculatorWidget

```
#include <QWidget>
#include <QLineEdit>
#include <QLabel>

class CalculatorWidget : public QWidget
{    Q_OBJECT
public:
    explicit CalculatorWidget(QWidget *parent = 0);
private slots:                                                   ①
    void valueChanged();
private:
    QLineEdit *m_firstValue;
    QLineEdit *m_secondValue;
    QLabel *m_result;
};
```

```
Listing 4-9  (continuation from previous page)
#include <QHBoxLayout>
#include <QFont>

CalculatorWidget::CalculatorWidget(QWidget *parent) :
   QWidget(parent)
{
    QHBoxLayout *layout = new QHBoxLayout(this);
    layout->addWidget( m_firstValue = new QLineEdit() );
    layout->addWidget( new QLabel("+") );
    layout->addWidget( m_secondValue = new QLineEdit() );
    layout->addWidget( new QLabel("=") );
    layout->addWidget( m_result = new QLabel() );
    QFont font = m_result->font();
    font.setBold(true);
    m_result->setFont(font);
    connect(m_firstValue, SIGNAL(textChanged(QString)),            ②
            this, SLOT(valueChanged()));
    connect(m_secondValue, SIGNAL(textChanged(QString)),           ③
            this, SLOT(valueChanged()));
    m_firstValue->setText("0");
    m_secondValue->setText("0");
}

void CalculatorWidget::valueChanged()
{
    QString firstValueText = m_firstValue->text();
    QString secondValueText = m_secondValue->text();

    bool first_ok,second_ok;
    int firstValue = firstValueText.toInt(&first_ok);
    int secondValue = secondValueText.toInt(&second_ok);

    if( first_ok && second_ok)
      m_result->setText(QString::number(firstValue+secondValue));
    else
      m_result->setText("no result");
}
```

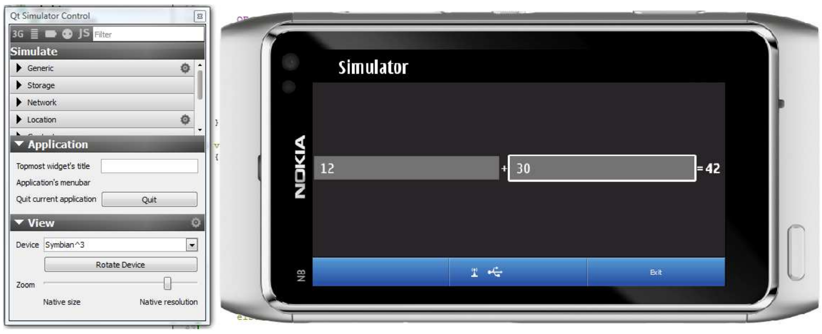


Figure 4-2 Visualization of running CalculatorWidget in Qt Symbian device simulator.

Take a look at lines ①②③ defining signals, slots and theirs interconnections. Signals and slots offer a way to communicate between objects. A signal can be emitted by any `QObject`. A slot is a member function that can be connected to particular signal, and which gets called in a response to the emitted signal. The signals and slots mechanism is more flexible since it doesn't require implementing interfaces and it offers loosely coupled many-to-many relationship.

Qt for Symbian port is adapted on top of Open C/C++ and native Symbian libraries (Qt base libraries are compiled as Symbian DLLs). S60 porting code implements the Symbian OS entry point `E32Main()` for Qt for Symbian and calls the standard `main()` entry point used by Qt. During the `QApplication` initialization, the Symbian application UI framework is initialized. The Nokia's SDK for Symbian can be used with either Qt Creator, or Carbide.c++. For our laboratory exercises, the most important is the fact that Qt for Symbian offers:

- Cross-platform application development without need to know the details of native Symbian C++.
- Qt Mobility APIs for taking advantage of mobile features.
- Advanced simulator for testing Qt programs for mobile devices which also supports Qt Mobility APIs (battery levels, charging states, profiles, storage, network connections, locations, contacts, messaging and sensors can be easily simulated) with different skin types including Maemo, Symbian Touch and Symbian NonTouch phones.
- Running and debugging the programs directly on Maemo or Symbian devices (there are debugging applications available for both platforms: App TRK for Symbian and Mad Developer for Maemo)
- On-device-debugging can be done by setting breakpoints and checking the variables.

Qt applications must conform to the Symbian OS security policies, just like any native Symbian programs. To enforce the security measures introduced in Symbian 9.x, a collection of software known as the Trusted Computing Base (TCB) is used. The TCB contains the kernel, the file system, and the software installer, and is responsible for ensuring that only applications with the necessary permissions and authority can be installed and are allowed to access restricted areas of the device.

Exercise SYM.ex4
Accessing Mobile Phone Features with Qt Mobility API

Innovate and implement any mobile application which utilizes Qt Mobility API to read/set mobile phone specific features (battery levels, locations, contacts, messaging or mobile device sensors). "Fall Detector" application, which detects if phone/person falls down and send an emergency email to predefined contact with current position, could be used as an example.
(Required Mobility APIs: Sensors – Acceleration, Location – GPS, Contacts - address book, Messaging - email)

Qt Mobility Project is a project that is creating a new suite of Qt cross-platform APIs for mobile devices functionality. It provides new APIs for: Sensors, Location, Messaging, Contacts, System information, Multimedia, etc. Successive new APIs are heavily developed as well: Camera, Telephony, Calendar. To use these APIs, a proper configuration definition has to be added to `*.pro` project file. For example, the following definition ①② should be added to inform about use of `location` and `systeminfo` API:

```
TEMPLATE = app
TARGET =
DEPENDPATH += .
INCLUDEPATH += .
CONFIG += mobility                                                    ①
MOBILITY = location systeminfo                                        ②
# Input
SOURCES += main.cpp
```

These APIs allow the developer to use features from one framework with ease and apply them to phones, netbooks and non-mobile personal computers. The framework not only improves many aspects of a mobile experience, because it improves the use of these technologies, but has applicability beyond the mobile device arena.

In the following subsections we briefly browse through short descriptions of available mobile API modules and illustrate their use with example code snippets:

System Information API

The System Information API provides a set of APIs to discover system related information and capabilities. Returned system information is related to a number of categories:

- Version - contains version information for a range of supporting software on the device. For example, from the Operating System and Firmware to the version of WebKit, Qt and the Service Framework.
- Features (hardware) - this lists the supported hardware on the device. Features include items such as the camera, bluetooth, GPS, FM radio etc.
- Network - the state of the network connection, and also the type of network e.g. gsm, cdma, ethernet etc.
- Display Information
- Storage Information - the presence of various storage devices. Including: none, internal, removable, cdrom.
- Device Information - Battery Status, Power State, Profile (silent, vibrating, normal etc), Sim, Input Method (key/buttons, keypad, qwerty, single touch screen, multitouch etc)
- Screensaver

A good illustration of `QSystemDeviceInfo` use is the following code snippet implementing visualization of device's battery state (in %) with `QProgressBar` widget. Every time, the system detects signal `batteryLevelChanged` the following signal-slot connection triggers the call of `setValue` slot function of `batteryLevelBar`.

```
// A code example: visualising the state of device's battery level

#include <QSystemInfo>
QTM_USE_NAMESPACE

void BatteryIndicator::setUp()
 {
     deviceInfo = new QSystemDeviceInfo(this);
     ui->batteryLevelBar->setValue(deviceInfo->batteryLevel());
     connect(deviceInfo, SIGNAL(batteryLevelChanged(int)),
             ui->batteryLevelBar, SLOT(setValue(int)));
 }
```

Messaging API

A unified interface for manipulation and storage of SMS , MMS , Email and XMPP messages It enables access to messaging services to search and sort messages, send messages, retrieve message data, and launch the preferred messaging client on the system to either display an existing message, compose a new message, or respond to an existing message. As an illustration we can use an example code which implements sending an email:

```
  // A Qt Mobility code example: sending an e-mail

  QMessage message;

  message.setType(QMessageAddress::Email);
  QString recipient("user@gmail.com");
  message.setTo(QMessageAddress(QMessageAddress::Email, recipient));
  message.setSubject("Qt Mobile API e-mail sending example");
  message.setBody("Example body text");

  QMessageService *m_service = new QMessageService();
  if (!m_service->send(message))
     QMessageBox::warning(0, "Failed", "Unable to send message");
```

Contacts API

The Contacts API allows developers to manage contact data in a platform-independent way. A contact consists of a set of contact details with own semantics of usage and storage with different context info (like separate phone number for work and home).

```
  // A Qt Mobility code example: Creating a new contact

  QContact Manager* cm = new QContactManager( this );

  QContact alice;                                // Create a new contact
  QContactName aliceName;
  aliceName.setFirst("Alice");
  aliceName.setLast("Jones");
  alice.saveDetail(&aliceName);

  QContactPhoneNumber number;      // Add a phone number to the contact
  number.setContexts(QContactDetail::ContextHome);
  number.setSubTypes(QContactPhoneNumber::SubTypeMobile);
  number.setNumber("12345678");
  alice.saveDetail(&number);
  alice.setPreferredDetail(”DialAction”, number);

  cm->saveContact(&alice);
```

Multimedia API

Multimedia provides a set of APIs that allow the developer to play, record and manage a collection of media content. There are several benefits this API brings to Qt. The developer can now implement fundamental multimedia functions with minimal code, mostly because they are already implemented. Also there is a great deal of flexibility with the media source or the generated multimedia. The source file does not need to be local to the device, it could be streamed from a remote location and identified by a URL. Finally, many different codecs are supported 'out of the box'. The Audio Player example is a good illustration to the basic use of the API:

```
// A Qt Mobility code example: Playing an audio file

QMediaPlayer *player = new QMediaPlayer( this );
...
player->setMedia(QUrl::fromLocalFile("audiofile.raw"));
player->setVolume(50);
player->play();
```

Sensors API

The Sensors API provides access to sensors. This covers both high-level sensors such as screen orientation (portrait, landscape) and low-level, real-time sensors such as accelerometer data. Sensor classes provide convenience wrappers that reduce the need for casting. Each of these classes represents a sensor type that the Sensors API knows about. Note that additional types may be made available at run-time. Currently, Sensors API supports following sensor classes:

- QAccelerometer (Linear acceleration along the X, Y and Z axes)
- QAmbientLightSensor
- QCompass
- QMagnetometer
- QOrientationSensor
- QProximitySensor (if something is close to the device)
- QRotationSensor (X, Y, Z –rotation of the device)
- QTapSensor (Registers tap and double tap events in 6 directions)

```
// A Qt Mobility code example: utilising device's orientation sensor

#include <QOrientationSensor>
#include <QPalette>
QTM_USE_NAMESPACE

class SensorWidget : public QWidget
{
    Q_OBJECT
public:
    SensorWidget(QWidget *parent = 0);
private slots:
    void onOrientationChanged();
};
```

```
static QOrientationSensor oSensor;

SensorWidget::SensorWidget(QWidget *parent) : QWidget(parent)
{
    connect(&oSensor,SIGNAL(readingChanged()),
            this, SLOT(onOrientationChanged()));
    oSensor.start();
}

void Widget::onOrientationChanged()
{
    QOrientationReading::Orientation orientation;
    orientation = oSensor.reading()->orientation();
    QPalette pal = palette();

    if ( orientation == QOrientationReading::TopUp ||
         orientation == QOrientationReading::TopDown )
       pal.setColor(QPalette::Background, Qt::blue);
    else
       pal.setColor(QPalette::Background, Qt::green);

    setPalette(pal);
}
```

Other currently supported Qt Mobile APIs are:

- **Qt Service Framework API** – which defines a unified way of finding, implementing and accessing services across multiple platforms.
- **Publish and Subscribe API** – which enables applications to read item values, navigate through and subscribe to change notifications.
- **Bearer Management API** – which controls the connectivity state of the system so that the user can start or stop interfaces or roam transparently between access points.
- **Location API** – which provides a library for distributing and receiving geographical location data using arbitrary data sources. The location info is obtained from satellite or other sources, e.g. GPS, Cell ID.
  Such location data usually involves a precisely specified position on the Earth's surface (provided by a latitude-longitude coordinates) along with associated data, such as:
  – the date and time at which the position was reported,
  – the velocity of the device that reported the position,
  – the altitude of the reported position (height above sea level),
  – the bearing of the device in degrees, relative to true north.

Detailed documentation of listed modules and for all the Qt Mobility Project APIs is included in Qt Creator local Help. It can be also found on page:
http://doc.trolltech.com/qtmobility-1.0/
Overview of the purpose and the main functionality of the Qt Mobility is described in the white paper: http://qt.nokia.com/files/pdf/qt-mobility-whitepaper-1.0.0
Since Trolltech acquisition, Nokia Company has been actively creating and sharing many e-learning Qt materials, which can be found on the "Qt in education" page:
http://qt.nokia.com/services-partners/qt-in-education/

5. Laboratory Class Schedule

The Mobile Computing course is planned on the last semester of the second level MSc diploma study. The auditory lecture, provides complete overview of fundamental paradigms, technologies and characteristic limitations related to mobile systems. The accompanying project will expand this overview, by individual studies on modern mobile trends including user focused and context-dependent services.

The objective of the laboratory class is to provide a real hands-on experience in software development for various hardware and programming environments, namely: Java Micro Edition, Windows Mobile, Symbian and Qt/QtMobility framework. This includes handling essential limitations of mobile environment: restricted user interface, limited computational performance, memory-awareness, power management and security in mobile applications. All laboratory participants are expected to posses theoretical and practical abilities suitable for fast creation of medium difficulty programs (about one hundred lines of code) in the time not exceeding one hour.

To speed up the pace of laboratory exercise, most of training exercises will have the form of small pre-prepared (pre-programmed) application frameworks, with some highlighted gaps (edit positions) to fill-in with the proper code. The assigned task, to be solved with the student's code, will be described in accompanying commentary. Knowledge of the theoretical concepts is required to solve these tasks. It is expected that the students have taken a look at the course slides and studied related chapters from this course book before the lab. Some “Repetition Forms” will be provided to repeat the theoretical foundations just before the class. According to tentative schedule, the laboratory will consists of following modules:

No	MODULE NAME / EXERCISE DESCRIPTION	TIMING
1	Java Microedition – Introduction Exercises: J2ME.ex1, J2ME.ex2	2h
2	Java Microedition – Continuation Exercises: J2ME.ex3, J2ME.ex4, J2ME.ex5	2h
3	Windows Mobile – Introduction Exercises: WM.ex1, WM.ex2	2h
4	Windows Mobile – Continuation Exercises: WM.ex3, WM.ex4, WM.ex5	2h
5	Symbian – Introduction Exercises: SYM.ex1, SYM.ex2	2h
6	Symbian with Qt / QtMobility framework Exercises: SYM.ex3, SYM.ex4	2h
7	Final presentation: teamwork implementation of selected multiplayer computer game (e.g. Asteroids or Star Wars) using wireless personal area networks (based on Bluetooth or WiFi) to enable collective game making use of many different mobile terminals accessible in the lab.	2h
8	Final assessment and examination	1h
	Total:	15h

Bibliography

[1] Coulton P., Edwards R., Clemson H., "S60 Programming: A Tutorial Guide", Wiley Publ., 2007.

[2] Fitzek F., Reichert F., "Mobile phone programming and its application to wireless networking", Springer, 2007.

[3] Ilyas M., Mahgoub I., "Mobile computing handbook", Auerbach Publ., 2005.

[4] Jipping, M. "Smartphone Operating System Concepts with Symbian OS", (Symbian Academy), Wiley Publ., 2007.

[5] Kroll M., Haustein S., "J2ME Application Development", Sams Publ., 2002.

[6] Knudsen J., Li S., "Beginning J2ME: From Novice to Professional", Apress Publ., 2005.

[7] Mikkonen T., "Programming mobile devices: an introduction for practitioners", Wiley Publ., 2007.

[8] Morris B., "The Symbian OS Architecture Sourcebook: Design and evolution of a mobile phone OS", Wiley Publ., 2007.

[9] Rischpater R., "Beginning Java ME Platform", Apress Publ., 2008.

[10] Stroustrup B., "The C++ Programming Language", Addison-Wesley Professional, 2000.

[11] Wigley A., Moth D., Foot P., "Microsoft® Mobile Development Handbook", Microsoft Press, 2007.

[12] VisionMobile Ltd, "Developer Economics 2010", report, 2010 http://www.visionmobile.com/blog/devecon/