Developing with IBM WebSphere Studio Device Developer

Stephen Pipes, IBM Hursley Park Labs, United Kingdom.

This lab will use IBM's WebSphere Studio Device Developer product to create an application that can be deployed and run on a Java-enabled device, such as a Personal Digital Assistant (PDA), handheld computer or phone. The lab has been taken from an IBM Redbook (see "Further reading" on page 36 for a link) and has been updated.
A basic understanding of Java programming is recommended before starting this lab. Also experience with the Eclipse environment will be very useful since Device Developer is based on Eclipse and provides a similar graphical user interface.

WebSphere Studio Device Developer 5.6 for Windows has been installed for this lab. A trial version for Windows or Linux can also be freely downloaded – go to http://www-306.ibm.com/software/wireless/wsdd/ for more details.

Introduction

This course will familiarise you with the embedded environment and show you how the WebSphere Studio family of tools helps you develop embedded Java applications. The sample application will be deployed in a software emulator on the development machine, although it can also be deployed on a handheld computer.

The embedded environment poses challenges to programmers, mostly due to the relatively limited capacities of the devices. The Java 2™ Micro Edition (J2ME) standards address this by defining standard Java packages that run on pre-defined hardware configurations. When deciding to support a specific J2ME configuration and profile, a hardware manufacturer will know what hardware is required to support the software. In addition, an application programmer will know what services are available to his application by looking at the configuration and profile specifications for a particular device.

Some Java knowledge is a requirement for doing the lab. The lab provides skeleton code for the exercises, but you will be writing Java code to complete them. We'll start with the basics of device programming and WebSphere Studio Device Developer, though. So, while experience in the other areas is advantageous, it is not necessary.
A device programming background is helpful, since you would then have some familiarity with the limitations of most devices, and how to best program in that environment. However, one advantage of the J2ME specifications is that application developers can assume support for whatever configuration and profile a device specifies. The standards function as a contract between the device manufacturer and the application developer, describing what functionality is available.

Lab objectives

Introduce WebSphere Studio Device Developer (herein referred to as Device Developer).
Understand the fundamentals of programming with J2ME.
Install a basic application and prepare the device emulator.
Develop the remaining parts of the application.
Deploy the application on the device emulator and test.

Throughout this lab, actions that should be performed by the reader begin with the  icon.

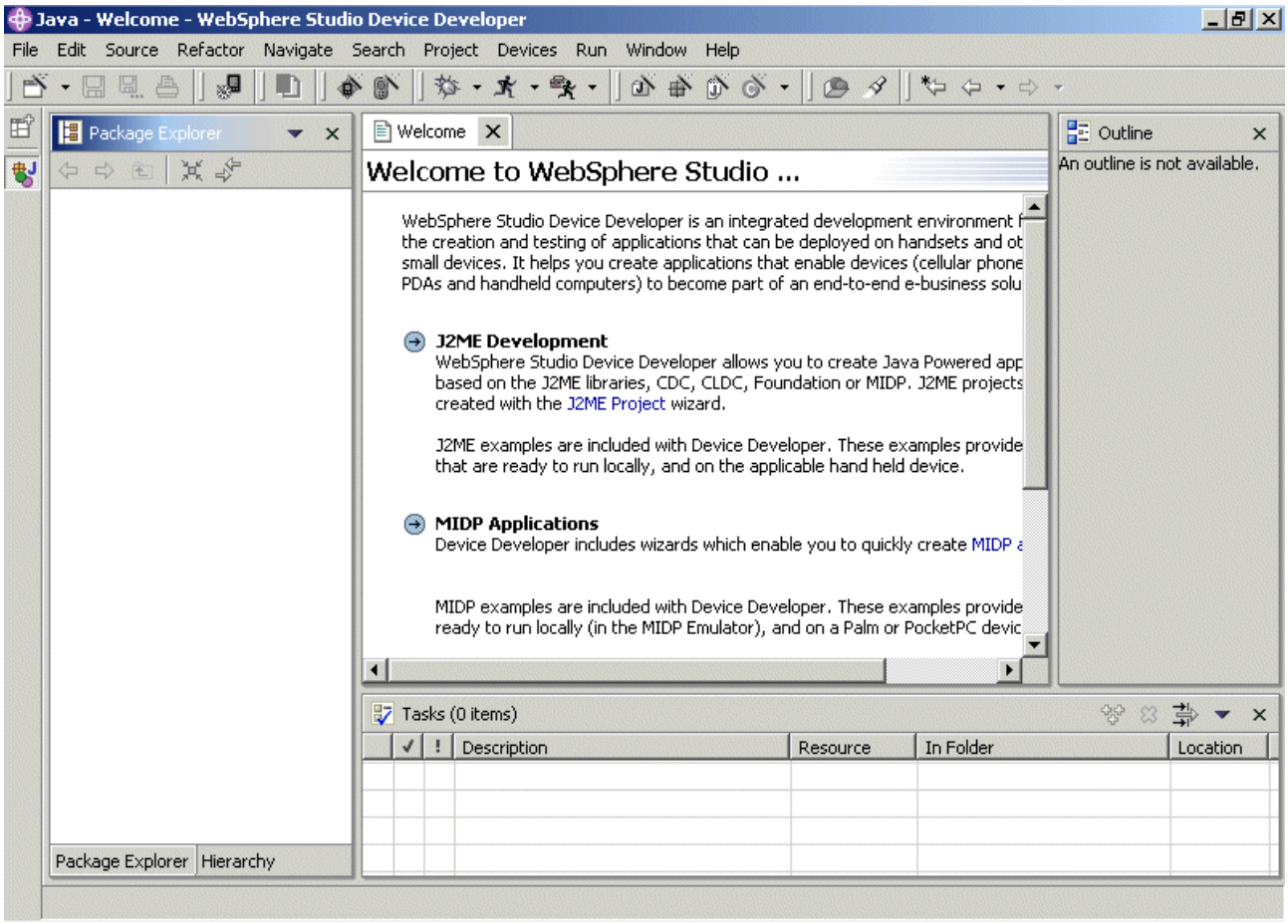
WebSphere Studio Device Developer

Device Developer is a platform for developing, debugging and deploying applications for resource-limited devices. However, as a member of the WebSphere Studio family, much of this may seem familiar if you have used other family members: WebSphere Studio Site Developer, WebSphere Studio Application Developer, and so forth. Also, these products are built on the Eclipse platform, so using vanilla Eclipse is another good way to become familiar with the Device Developer basic functionality.

The basic Integrated Development Environment also ships with a copy of the WebSphere Micro Environment (IBM J2ME-compliant JVM), licensed for development use.

When installed, Device Developer creates an icon on the desktop.

Double-click this icon to load Device Developer.
The following screen should appear on the desktop.



These screenshots were taken from a Windows 2000 machine. Although the images will look slightly different on a Windows XP machine, the functionality is the same.

The top toolbar (under the menu bar) contains all of the available Device Developer wizards, and will change to match the current perspective. Wizards are available to create applications, define

devices, test code and create skeleton Java structures (classes, interfaces, and so on) within your programs.
The left-hand toolbar contains all of the currently open perspectives and/or fast views. As you open a Perspective, its icon will be added to the left-hand toolbar. To close a Perspective when you are finished with it, or when there are too many opened, simply right-click on its icon. From the context menu that appears, you may close that perspective or all open perspectives. To create a fast view, simply drag any view to the toolbar. It will close and its icon will appear in the toolbar. To re-open it, double-click the icon.

A Perspective is a collection of Views and Editors designed to focus on a particular task. The Java Perspective is focused on writing Java code, the Debugging Perspective is for debugging code, and so on. You can also customise Perspectives and save them, if the default layouts are not to your liking.

A View is just that: a view of something in the workspace. For example, an Outline view is a view of a ".java" file broken up into an outline format.

An Editor is used to modify files, and is specific to the file type being edited. If you do not like the default editors provided with Device Developer, you can specify external editors for specific file types. Only one editor can be active at any time – multiple editors can be opened, but they will all be in the same window, available via tabs. To change editors, click the tab of the editor you wish to make active.
If you change something in an editor, you will generally be prompted to save it before it is used (before executing code, for example). You can set an option in the Preferences to automatically save a file before it is used. If a file has been changed in an Editor and not saved, the tab will contain an asterisk by the file name to remind you that the contents need to be saved.

The default perspective is "Java" which displays the Package Explorer, Welcome, Outline and Tasks views. The Hierarchy view is also available by clicking the "Hierarchy" tab but is not displayed by default.

All code is stored in separate files in a "workspace", a directory-based repository that defaults to the workspace directory within the Device Developer installation directory. The appropriate workspace is selected as Device Developer starts.

WebSphere Everyplace Micro Environment

WebSphere Everyplace Micro Environment is a Java Powered Runtime Environment that provides the foundation for the deployment of e-business applications to small mobile devices.

From the IBM Web site:
"A major, integrated component of the IBM Workplace Client Technology, Micro Edition platform, which packages WebSphere Studio Device Developer with WebSphere Everyplace Micro Environment to create an enhanced offering that can assist in the development, testing, and deployment of applications to server-managed clients.
Contains a production-ready Java Powered runtime environment, tested and certified to meet J2ME™ specifications as laid out by the Java Community Process™. WebSphere Everyplace Micro Environment supports:

- Connected, Limited Device Configuration (CLDC 1.0 and 1.1) and Mobile Information Device Profile (MIDP 2.0) for the palmOne Tungsten C and HP iPaQ Pocket PC h5550.
- Connected Device Configuration (CDC 1.0_01), Foundation Profile, and Personal Profile for the HP iPaQ Pocket PC h5550 and the Sharp Zaurus SL-6000.

The use of industry standards and middleware enables devices to be connected to many existing e-business applications. By testing middleware and server connectivity, WebSphere Everyplace Micro Environment combines the convenience of mobile devices with the power of e-business. When used with other middleware products, many existing enterprise applications can be extended to server-managed mobile and pervasive devices:

- Web services (SOAP) support can help enable access to applications across a wide variety of wireless and wire-line networks.
- Testing with assured messaging software (MQe, JMS) can provide access to assured messaging and financial transactions.
- When used with DB2e, the advanced data management capabilities of DB2 can also be leveraged. Applications, middleware, and runtimes can be server-managed when used in conjunction with IBM SMF".

Refer to http://www-306.ibm.com/software/wireless/weme/ for more details.

WebSphere Everyplace Custom Environment

This component of Workplace Client Technology, Micro Edition provides the ability to use custom profiles (i.e. those not defined in J2ME) for applications where footprint and performance are of the most importance. WebSphere Everyplace Custom Environment provides a complete runtime environment for the deployment of embedded applications to real-time control systems, automotive telematics systems, and other deeply embedded applications.

Refer to http://www-306.ibm.com/software/wireless/wece/ for more details.

Java 2 Micro Edition

J2ME™ is designed for small devices with limited capacity and functionality. Small devices range in size from pages, mobile phones and PDAs, all the way up to things like set-top boxes and vehicle telematics. The device must be capable of running a Java Virtual Machine, such as those developed by IBM.
J2ME consists of **configurations**, **profiles** and **optional APIs**, which provide specific information about APIs and different families of devices. A configuration is designed for a specific type of device and is based on memory constraints and processor power. A Java Virtual Machine (JVM) supporting the required configuration and a subset of Java API's for this configuration are specified here.
Profiles are more specific; they are based on a particular configuration and add APIs for user interface, network connectivity, data storage, and everything else that is necessary to develop running applications.
Optional APIs define specific additional functionality that may be included in a configuration.
These three properties are combined to form a **stack**, e.g. CLDC+MIDP+Wireless Messaging API.

A number of configurations and profiles already exist today.

<table>
<tr><td colspan="3"><- Smaller ---</td><td colspan="3">--- Larger -></td></tr>
<tr><td>Pagers</td><td>Mobile phones</td><td>PDAs</td><td>Car navigation</td><td>Internet appliances</td><td>Set-top boxes</td></tr>
<tr><td colspan="2" rowspan="3">MIDP (Mobile Information Device Profile)</td><td colspan="2" rowspan="3">PDAP (Personal Digital Assistant Profile)</td><td colspan="2">Personal Profile</td></tr>
<tr><td colspan="2">Personal Basis Profile</td></tr>
<tr><td colspan="2">Foundation Profile</td></tr>
<tr><td colspan="4">CLDC (Connected, Limited Device Configuration)</td><td colspan="2">CDC (Connected Device Configuration)</td></tr>
<tr><td colspan="6">J2ME (Java 2, Micro Edition)</td></tr>
</table>

The Java Community Process (JCP) is designed to ensure that Java technology is developed according to community consensus. See http://jcp.org/introduction/overview. Configurations and profiles are initiated as Java Specification Requests (JSRs). The current list is at http://jcp.org/jsr/all.

This lab will focus on CLDC+MIDP which is intended for mobile devices with extremely limited resources. Although the name implies the device should be permanently connected, this is not necessary and in fact many mobile devices will only be able to manage intermittent connectivity. In addition to this, network throughput will probably be very low. A basic GSM connection may only offer 9.6 Kbps, for instance.

According to the specification, a MIDP device has the following characteristics:

- 128KB of non-volatile memory for the MIDP implementation
- 32KB of volatile memory for the runtime heap
- 8KB of non-volatile memory for persistent data
- A screen of at least 96x54 pixels
- Some capacity for input, either by keypad, keyboard or touch screen
- Two-way network connection, possibly intermittent.

Device Developer supports MIDP and provides tools to start building an application quickly.

Developing a MIDP application

The lab will focus on using Device Developer to develop MIDlets (which is shorthand for MIDP applets) and making a selection of GUI classes within the MIDP API. If you have some experience writing Java applications, then the transition to MIDlets should not be difficult. There are some extra classes provided with the profile but the language is still Java, as it has always been.
It is also assumed that you have some experience with the Eclipse development environment, in which case you should have little trouble in recognising the Device Developer Graphical User Interface and understanding its functions.

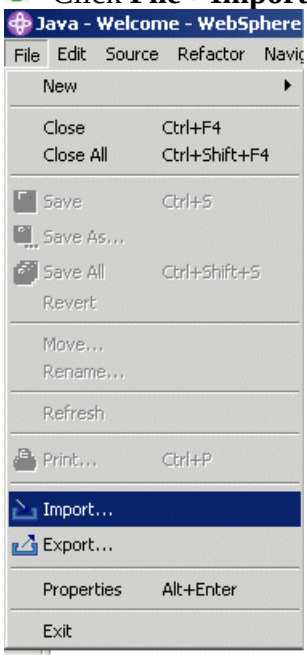
This lab develops an application that serves as stopwatch. The application will have start/stop buttons to start the stopwatch and an exit button to terminate. Users have the choice of selecting various display modes for the stopwatch. These modes are:

- Minutes
- Seconds
- Milliseconds
- Roman Numerals

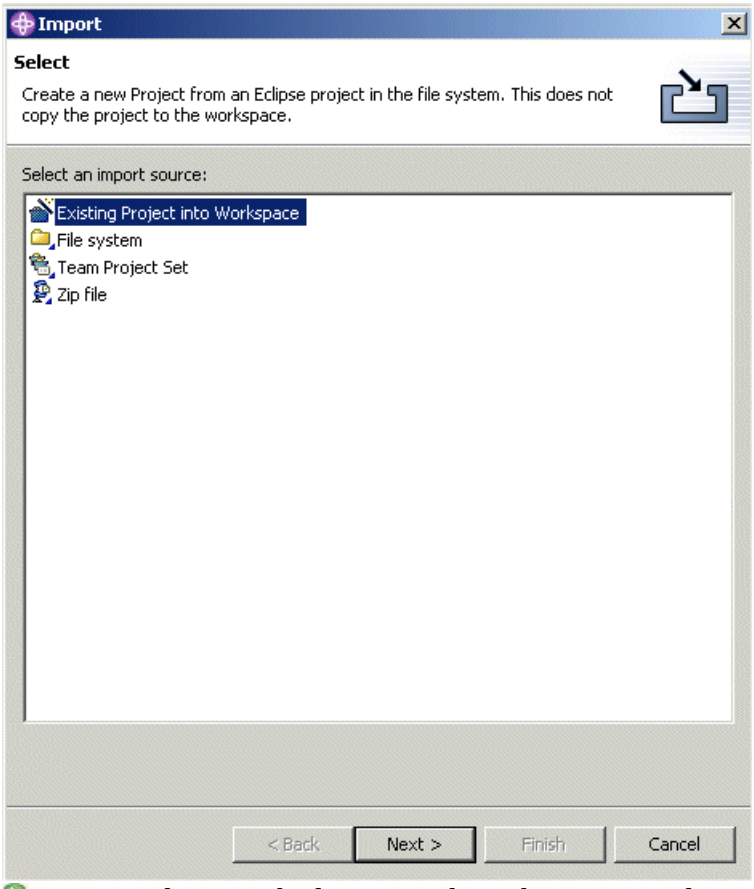
This lab will explain the following tasks:

- Importing a Project
- Creating a MIDP Form
- Creating a Command Listener
- Creating a Display Thread
- Running the MIDP Application Locally
- Running the MIDP Application on the Device

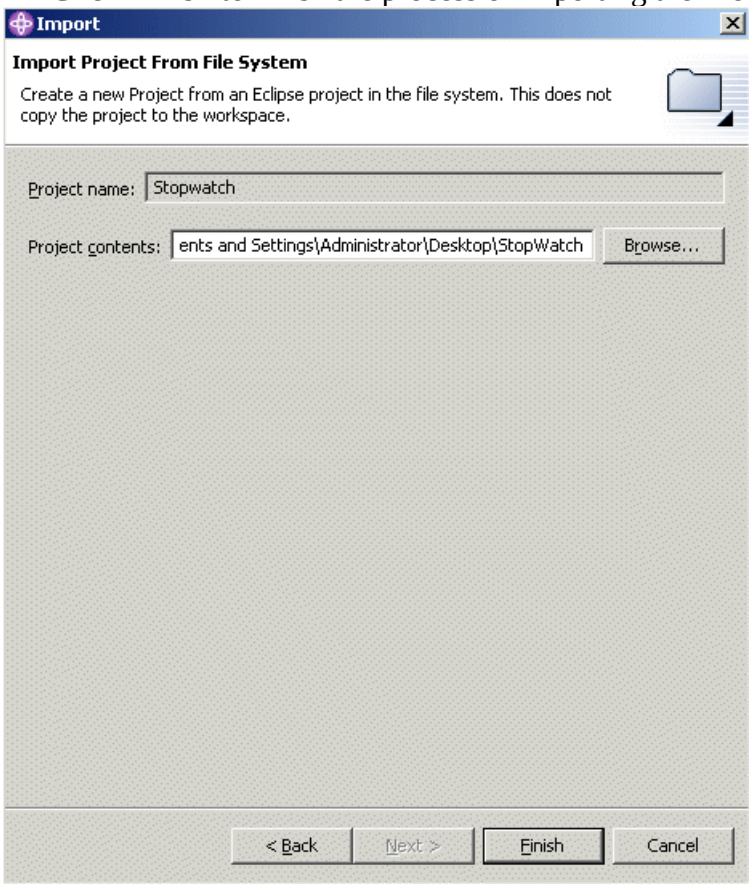
- Click the **Java Perspective** icon in the left-hand corner of the Device Developer Window.
- Click **File->Import** to start the process of importing a project into the current workspace.



- Specify an import source by selecting **Existing Project into Workspace** in the import window and then click **Next**.

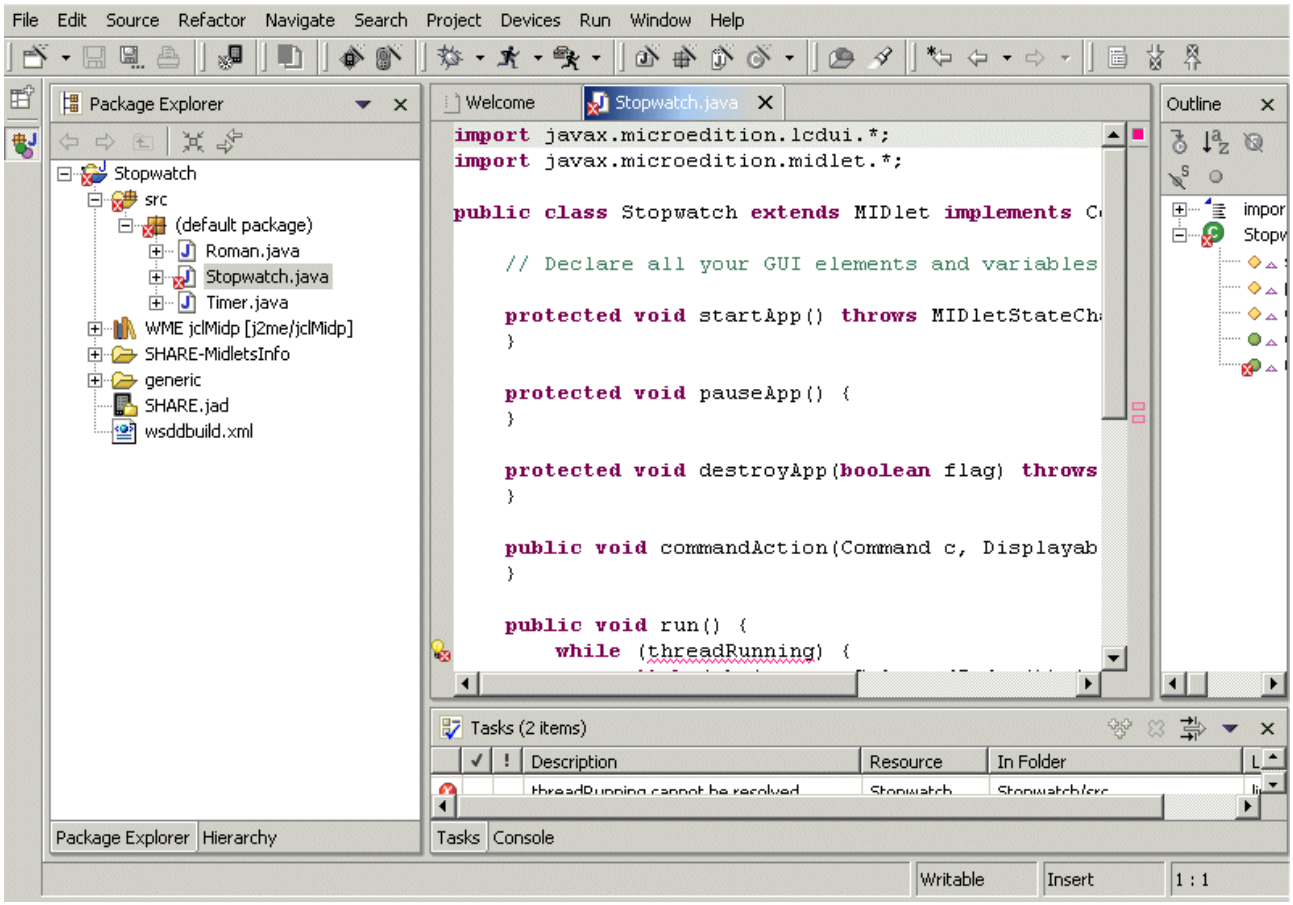


- Browse down to the location where the project is located. The instructor will provide details on where the Stopwatch lab is located.
- Click **Finish** to finish the process of importing the file into the workspace.



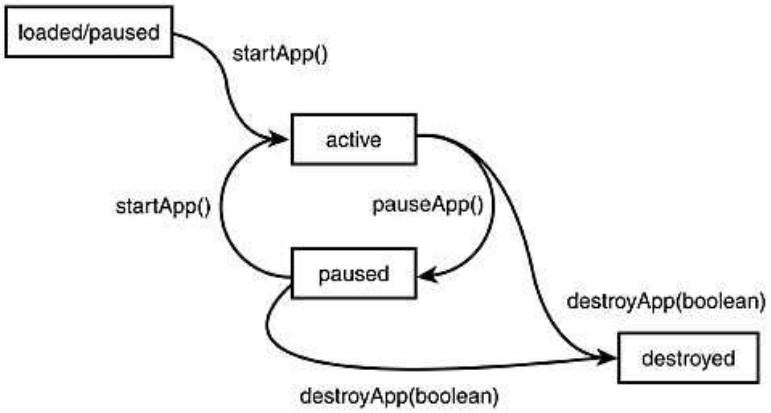
Now, the MIDP form can be created.

1. In the Package Explorer, expand the new **Stopwatch** project (this can be achieved by clicking the ⊞ icon to the left of "Stopwatch"). Then expand the **src** folder, then the **default package** folder. You will notice that there are three Java files which need to be completed before this application can be run.
2. Double-click the **Stopwatch.java file** to open the Java editor. Currently the file is just a skeleton. Notice that the class extends MIDlet (and is not abstract), which informs the Java compiler that it will implement the **startApp()**, **pauseApp()** and **destroyApp(boolean)** methods. The class also implements the **CommandListener** and **Runnable** interfaces, which informs the compiler that the **commandAction(Command, Displayable)** and **run()** methods will be implemented.



The MIDlet lifecycle

MIDlets have a specific lifecycle which are implemented in the methods shown in the sample code. The installation and operation of the MIDlet is controlled by the device (or device emulator) – installation consists of moving the Java classes (contained in a JAR or JXE archive, see later for details) to the device, while an associated descriptor file (with a .jad extension) describes the archive contents.
According to the specification, a MIDlet can be in one of the following states at any particular time.

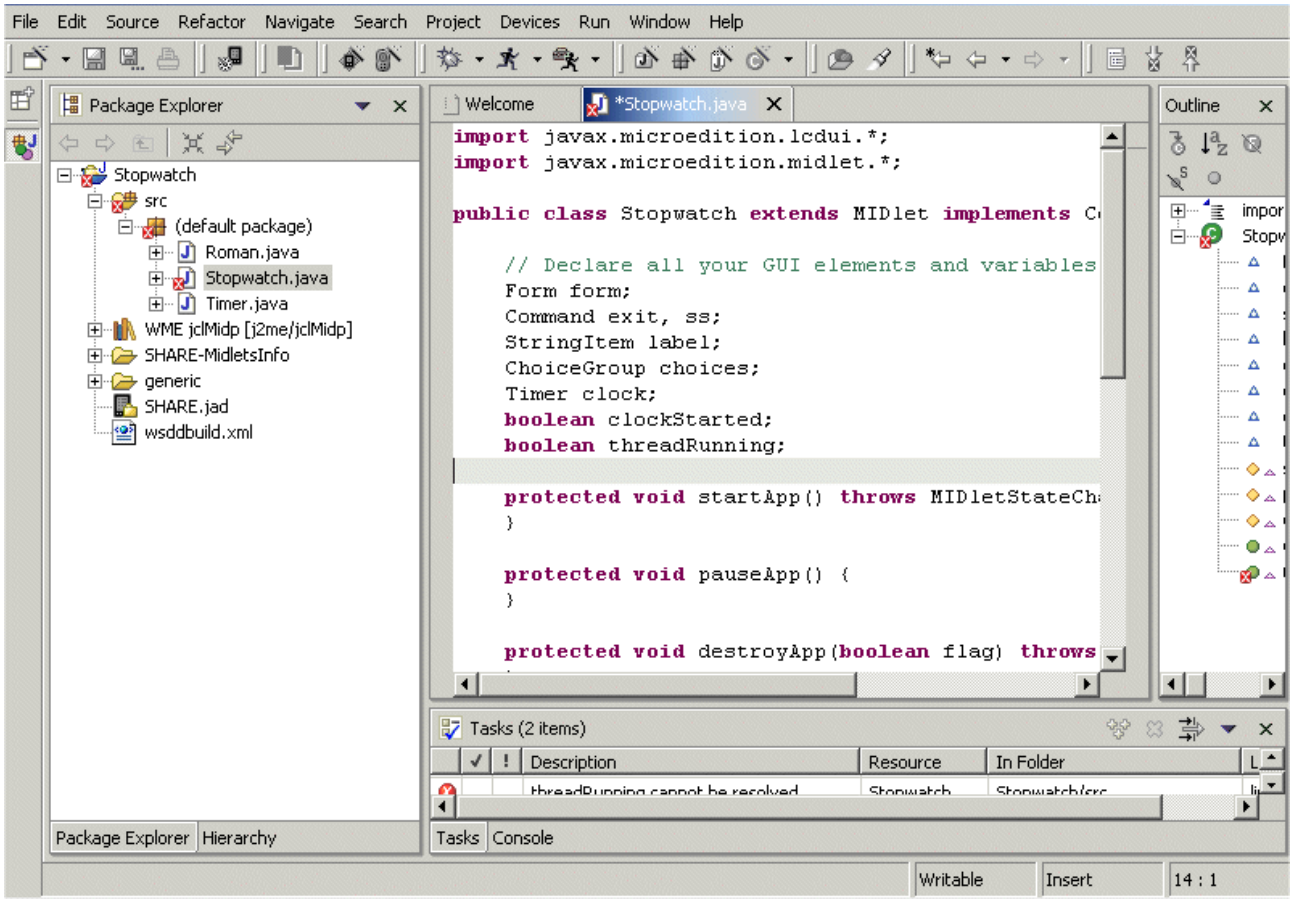


1. When the MIDlet is about to be run, an instance is created. The MIDlet's constructor is run, and the MIDlet is in the **Loaded/Paused** state.
2. Next, the MIDlet enters the **Active** state after the device calls startApp().
3. While the MIDlet is Active, the device can suspend the application by calling pauseApp(). This puts the MIDlet in the **Paused** state. A MIDlet can place itself in the Paused state by calling notifyPaused().
4. The device can terminate the execution of the MIDlet by calling destroyApp(), at which point the MIDlet is **Destroyed** and is eligible for garbage collection. A MIDlet can destroy itself by calling notifyDestroyed().

Creating a MIDP form

- First, several “class-level” variables must be defined. These will indicate to the compiler what types of objects are required, their visibility and names. Below the comment “Declare all your GUI elements and variables here” (but above the first method), enter the following declarations.

```
Form form;
Command exit, ss;
StringItem label;
ChoiceGroup choices;
Timer clock;
boolean clockStarted;
boolean threadRunning;
```



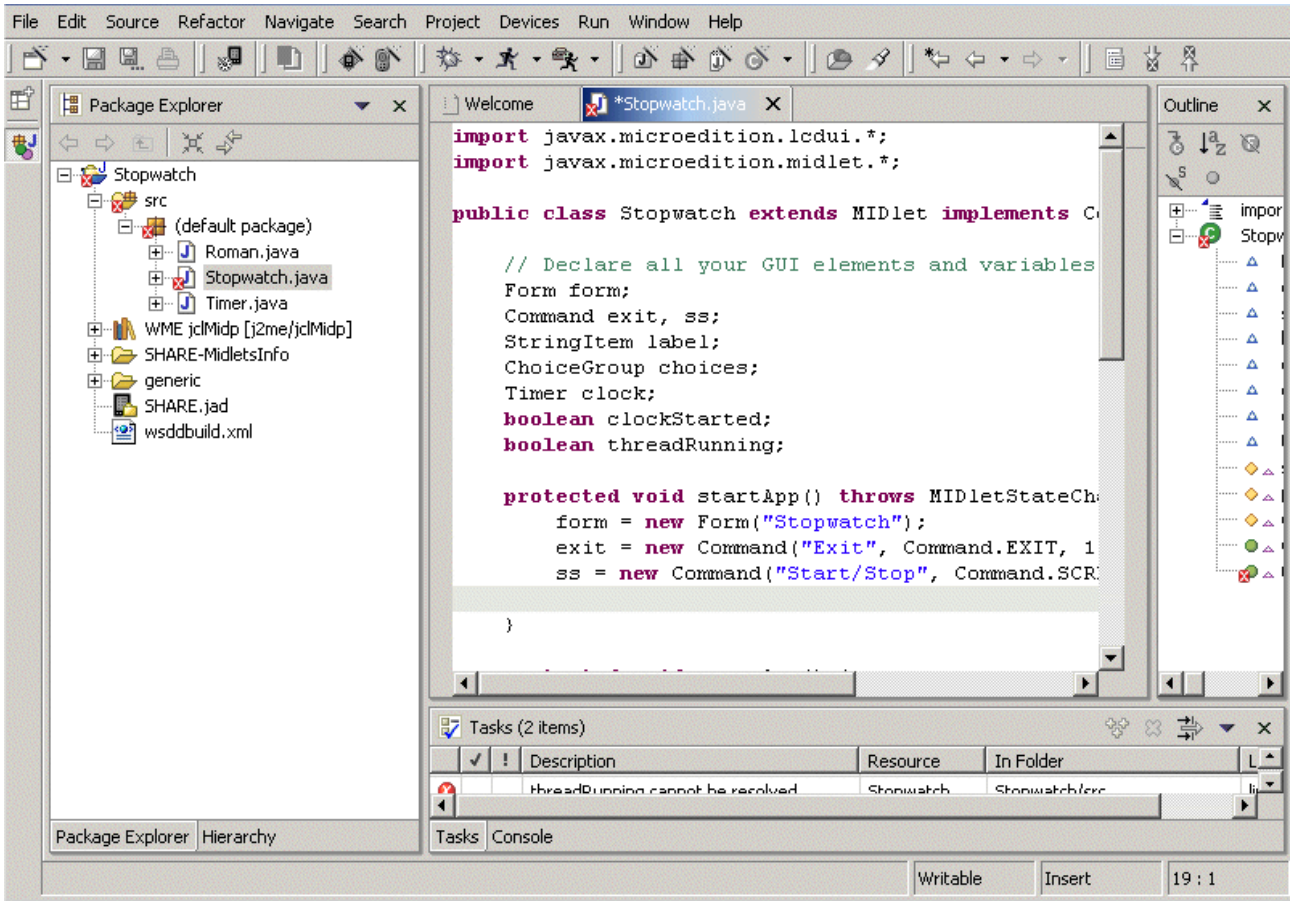
5. Inside the **startApp()** method, create a Form (**javax.microedition.lcdui.Form**) object. A Form object acts as a container for GUI elements. The constructor for Form takes a single String object as a parameter. This String becomes the title of the Form.

```
form = new Form("Stopwatch");
```

6. Immediately after the code above, create two **Command** (**javax.microedition.lcdui.Command**) objects. A Command object represents "Soft Buttons" that are placed according to device rules.

```
exit = new Command("Exit", Command.EXIT, 1);
ss = new Command("Start/Stop", Command.SCREEN, 2);
```

If you want to learn more about what the Command parameters mean, then place the cursor over the word **Command** in the editor. A window should pop up with more information.



Device Developer will automatically compile Java code as it is edited. There is no need to manually perform the compilation step. If you have entered the code incorrectly, the compiler may report an error, which will need to be resolved before the class can be instantiated during runtime. If the compiler spots an error, an error icon will appear in the left hand margin of the Java editor view. You can place the cursor over this icon and a pop up window should explain why compilation has failed at that point.
If there are no compilation errors, then you should see no error icons!

7. Following the code just entered, add the **StringItem** code below so that we can display some text in our application. Again, more information can be found by placing the cursor over **StringItem**.

```
label = new StringItem("Stopwatch","0 min.");
```

8. After the StringItem code, a **ChoiceGroup** (**javax.microedition.lcdui.StringItem**) object needs to be created. A ChoiceGroup allows the selection of choices, like a set of radio buttons or checkboxes.

```
choices = new ChoiceGroup(null,Choice.EXCLUSIVE);
```

9. Next, choices need to be added to the **ChoiceGroup**. Choices are added using the **insert(int,String,Image)** method. The integer is the sequence of the choice, the **String** is the name of the choice, and the **Image** (in this case **null**) is displayed next to the choice.

```
choices.insert(0, "minutes", null);
choices.insert(1, "seconds", null);
choices.insert(2, "milliseconds", null);
choices.insert(3, "Roman Numerals", null);
```

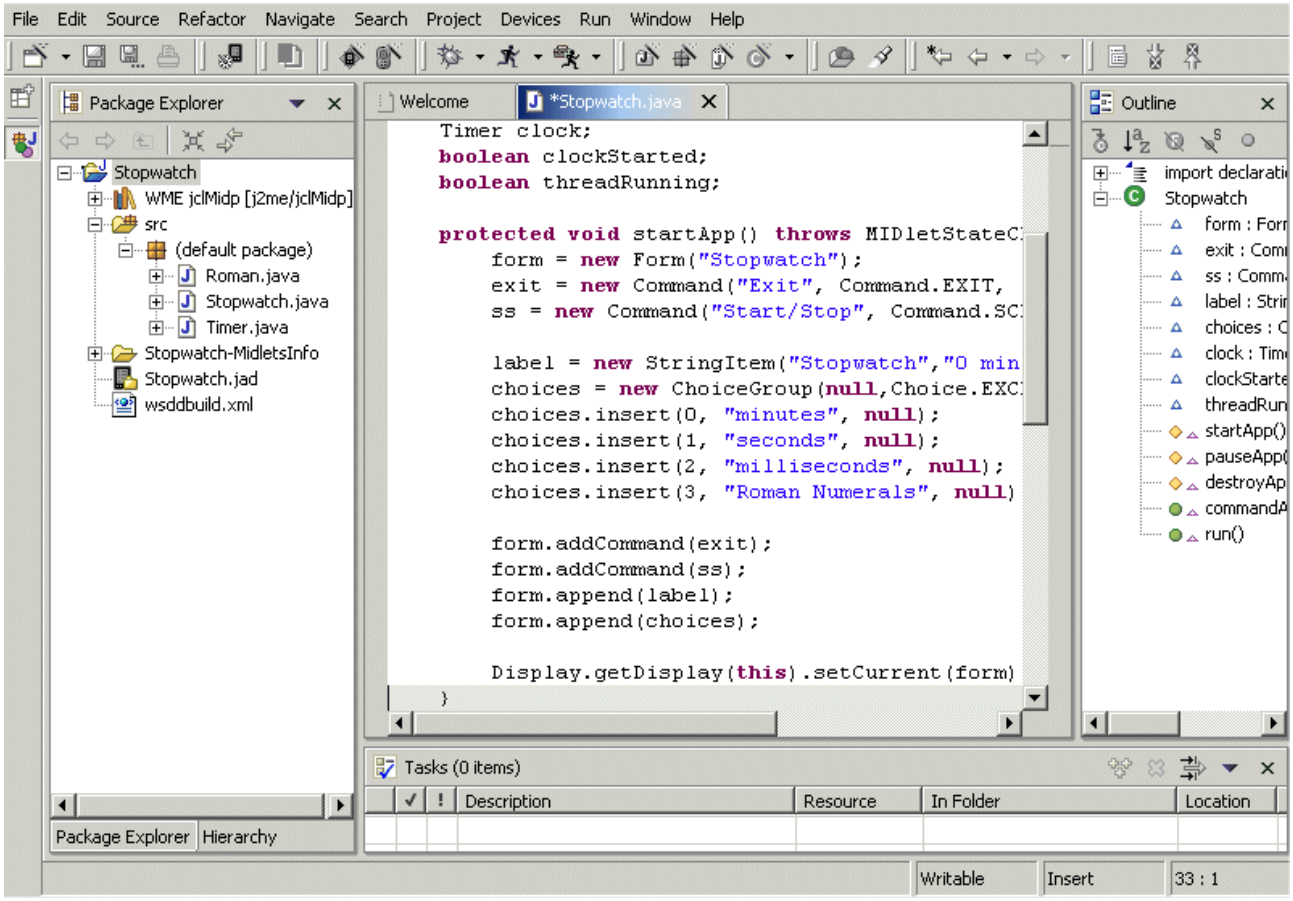
10. Now that all of the GUI elements have been created, they need to be added to a container, which in this lab is the Form. Use the Form's **addCommand(Command)** method to add the two **Command** objects and the **append(Item)** method to add the **ChoiceGroup** and the **StringItem**. Both ChoiceGroup and StringItem extend the Item class.

```
form.addCommand(exit);
form.addCommand(ss);
form.append(label);
form.append(choices);
```

11. The last part of creating a form is to display it. First, a reference to the **Display** object needs to be retrieved using the static **Display.getDisplay(MIDlet)** method. Since the Stopwatch class extends the MIDlet class, a "**this**" pointer can be passed as a parameter. Then, use the **Display** object's **setCurrent(Displayable)** method to display the Form. Form extends the **Displayable** class.

```
Display.getDisplay(this).setCurrent(form);
```

The form is now complete. Your code should look like this:



It is the responsibility of the JVM running on the device to format this form correctly. How the form is eventually displayed will depend on the capabilities of the device's display. We will not assume specific capabilities when developing our application; instead we will develop code that will run on any hardware that supports MIDP.

Creating a command listener

Before the application can respond to events, such as a button press, a **Listener** must be created. For this lab, the listener will be the Stopwatch class. Notice that Stopwatch implements the

CommandListener Interface. This interface declares a single method, **commandAction(Command, Displayable)**. The Command parameter is the Command that was activated, and the Displayable parameter is the object that "heard" the event.

Within the **commandAction(Command, Displayable)** method, there needs to be logic to handle the event if the Command labeled "exit" was activated. The Command method parameter should be compared to the Command object that was labeled "exit". If this was the Command that was activated, the **destroyApp(boolean)** method should be called and then the **notifyDestroyed()** method. Note that the **destroyApp(boolean)** method throws an exception and must be placed in a try/catch block.

12. If the Command labelled “exit” is activated, then the application should close down. If “start/stop” is activated, then the clock should respond accordingly. In the **commandAction(Command, Displayable)** method, enter the following code.

```
if (c == exit) {
   try{
      destroyApp(true);
      notifyDestroyed();
   } catch(Exception e){}
}
else if (c == ss) {
   if (clockStarted) clock.stop();
   else clock.start();
   clockStarted = !(clockStarted);
}
```

13. In the **startApp()** method, create an instance of the Timer class. A variable is also needed to determine if the Timer object is started.

```
clock = new Timer();
clockStarted = false;
```

14. The final step is to add the CommandListener to the previously created form. Add the following code to the **startApp()** method.

```
form.setCommandListener(this);
```

Creating a display thread

Even though the application can now respond to events, the display does not reflect this. To update the display, a separate **Thread** will be used. Notice that the Stopwatch class implements the **Runnable** interface. This means that Stopwatch must have a **run()** method. This is the method that will be executed when a new Thread starts.

15. Initialise the **threadRunning** variable and start the thread running by adding the following code to the **startApp()** method.

```
threadRunning = true;
new Thread(this).start();
```

The **run()** method should execute in a loop until the display thread state variable changes from "running" to "stopped". Inside the loop the **StringItem** object should be updated with the results of the Timer object's **read()** method. The **read()** method returns a long number representing the number of milliseconds the Timer object has been running. This can be mathematically adjusted to show seconds and minutes as well. The Roman class' static **toRoman(int)** method can be used to

convert any number to roman numerals. To determine which number format the user selected from the **ChoiceGroup** object, use the object's **getSelectedIndex()** method. This method returns the sequence number of the user selected choice.

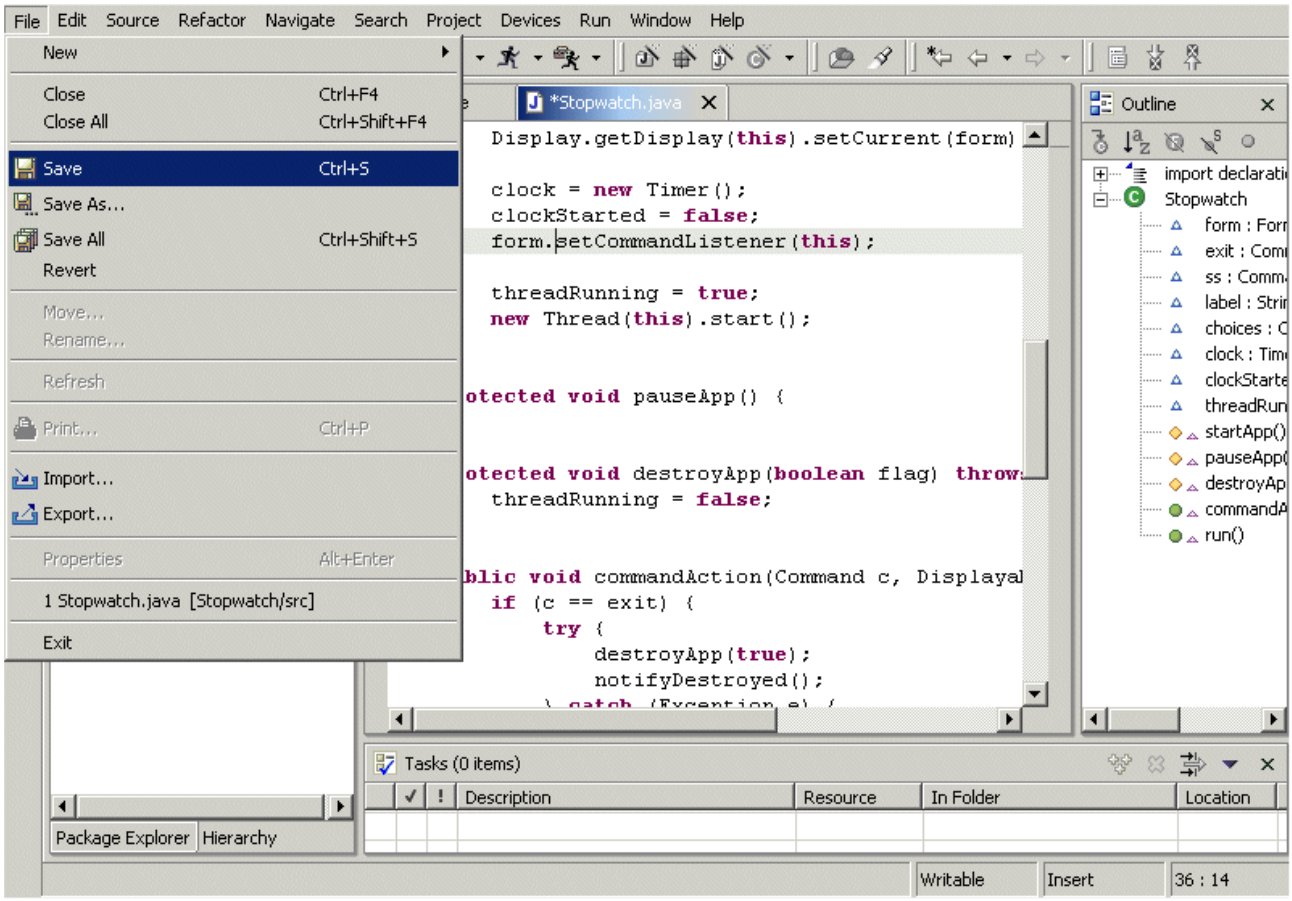
16. Verify that the following code is already present in the **run()** method. (It should have been added beforehand to save time during the lab).

```
while (threadRunning) {
      switch(choices.getSelectedIndex()) {
      case 0 :
            label.setText(clock.read() / 60000+ " min.");
            break;
      case 1 :
            label.setText(clock.read() / 1000+ " sec.");
            break;
      case 2 :
            label.setText(clock.read() + " ms.");
            break;
      case 3 :
            label.setText(Roman.toRoman((int) clock.read()));
            break;
      }
}
```

17. The last step is to verify that the Thread stops when the MIDlet ends. Add the following code to the **destroyApp(boolean)** method.

```
threadRunning = false;
```

- Finally, save the new code by selecting **File** -> **Save** from the menu bar.



Running the MIDP application on the development machine

It is often quicker to test an application in the development environment before deploying to a device. Thanks to Device Developer's built-in debugger, a MIDP device is not actually required to detect and repair Java errors introduced during development.

The first step is to create a runtime environment for our new MIDlet. A **MIDlet suite** is a requirement for the runtime environment. According to the Device Developer documentation, this is a JAR file containing one or more MIDlets plus the following:

A runtime execution environment
MIDlet suite packaging (that is, a manifest, Java class files, and resource files)
A Java Application Descriptor (JAD) file
Application life cycle (for starting, stopping, and cleaning up applications).

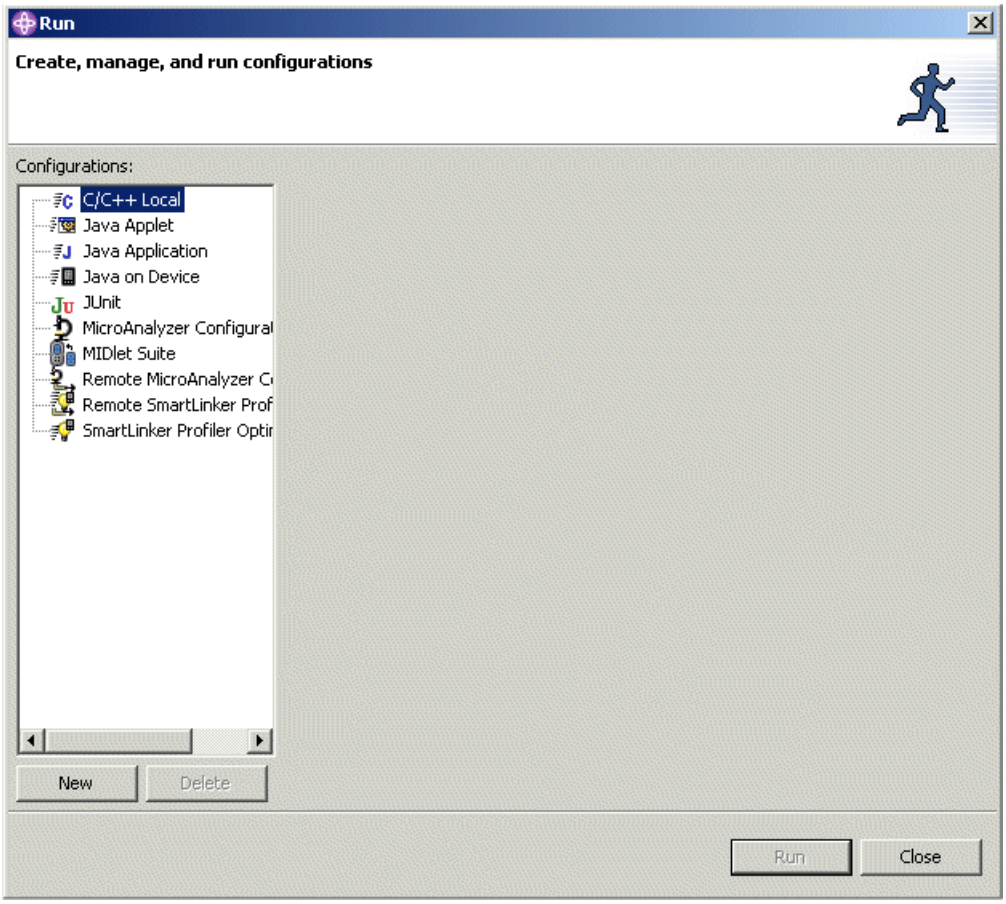
The JAD file describes a MIDlet or a MIDlet suite (multiple similar midlets) in text format. It must contain, at the very least, information about version, vendor, target-analyze and target-configuration.

JAD files do the following:

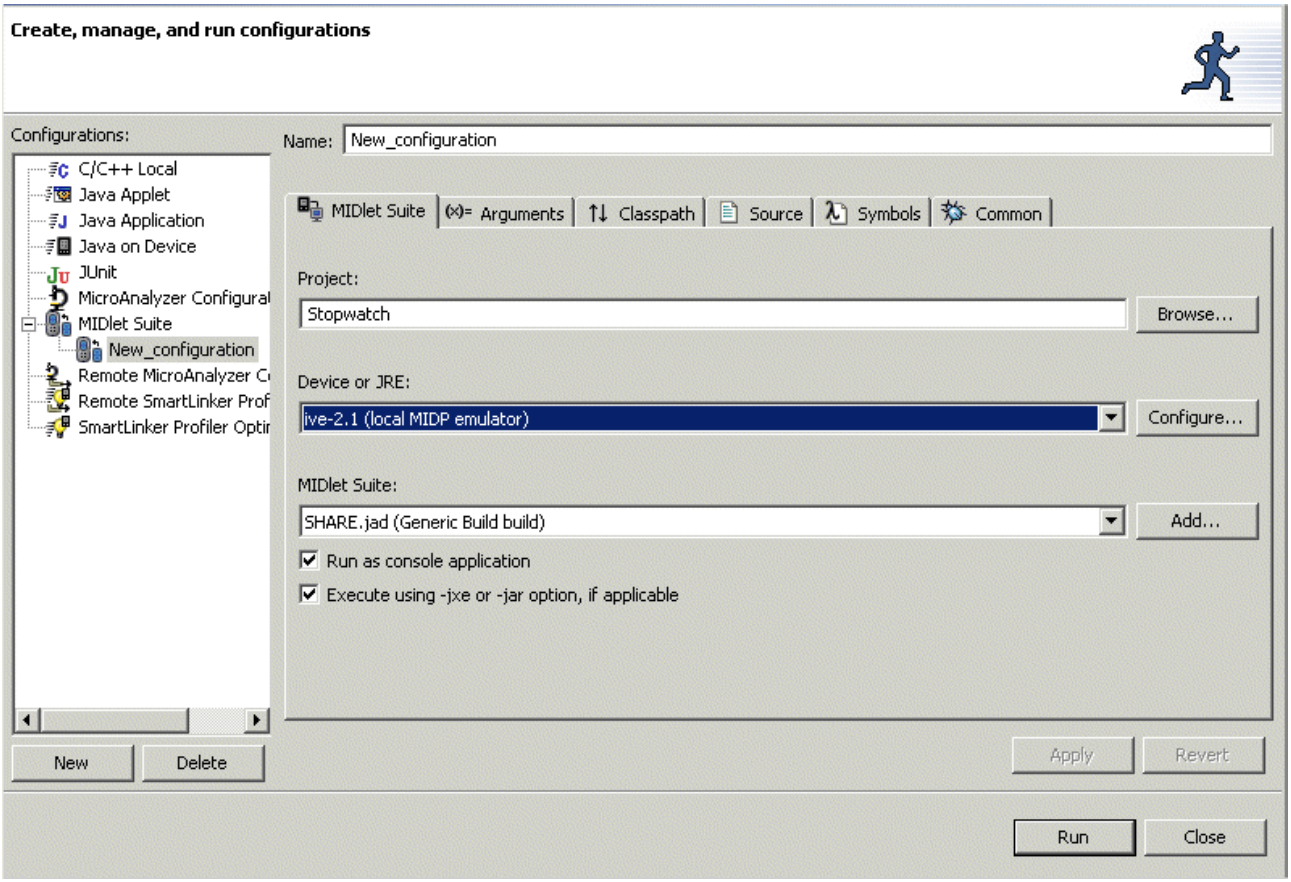
Identify the MIDlets contained within in a JAR file
Specify information (both required and optional) about these MIDlets
Specify the size of the JAR file that contains the MIDlets
Optionally, a JAD file can also include parameters to pass to the MIDlets in the JAR file at runtime
If building a MIDlet suite, a JAD file must be included to specify the names of all of the MIDlets in the suite. All of the files in a MIDlet suite can also be run from the JAD file that describes its contents.

Fortunately, a MIDP 1.0 MIDlet suite has been prepared, so this step is not necessary in this lab. The next stage is to create a runtime **configuration** for our MIDlet.

- Select **Run** -> **Run…** from the menu bar. The **Launch Configurations** window will appear.



- Select **MIDlet Suite** in the **Configurations** list and click **New**. This will create a new runtime configuration.
- Ensure that the following information is present. The **Project** text area should contain "Stopwatch" (without the quotes), the **Device or JRE** drop-down list should contain "ive-2.1 (local MIDP emulator)" and the **MIDlet Suite** drop-down list should contain "SHARE.jad (Generic Build build)". This is the name of the predefined MIDlet suite. **Run as console application** and **Execute using –jxe or –jar option, if applicable** should be checked.



- Click **Run**.

After a short while, the emulator should appear. During this time, Device Developer adds some additional runtime information to the MIDlet suite defined in the Stopwatch project. More information on this in the section “Building and Launching on Devices” on page 18.

- Test the application in the MIDlet emulator by selecting a unit time and pressing the **Start/Stop** button on the phone (the button is located below the phone’s display – if you click anywhere on the phone’s display then nothing with happen). This emulator can be used to test every aspect of a MIDP application before deploying on a real device and is a great time-saver during development.

Building and Launching on Devices

WebSphere Studio Device Developer supports deploying code to multiple devices. Each device is defined and then as projects are created, a **launch** is prepared to run the project on a specific device. Within each project there is the option to build code for a specific platform or for multiple platforms. Once the code is has been built, it is launched on a specific device (either a real device or through an emulator).
The section just completed demonstrates how to build for a MIDP device and Device Developer also directly supports Palm devices, the Palm emulator and PocketPC devices, as well as desktops running Windows and Linux. The complete list of supported devices can be seen in the **Device Configuration** window, which may be accessed from the menu bar under **Devices** -> **Configure…**. Other devices may be supported through plug-ins from the device manufacturer.

Building an application

A **build** is a version of the source code compiled to run on a specific device. A **device** is a test platform, either real or emulated. A **launch** ties a build to a specific device – it runs that build on the selected device.
Builds and launches can be created for each project. Devices are defined once for all projects in Device Developer.

Ant ("Another Nice Tool") is an XML-based make facility for Java. Its use is integrated into Device Developer and can be used without knowledge of the underlying Ant syntax. None-the-less, a competent Ant programmer can customise the scripts generated by Device Developer.

Here are the four steps to test a project:

1. Create a build.
2. Create a device profile.
3. Create a launch configuration that uses the build and device profile.
4. Run the launch configuration.

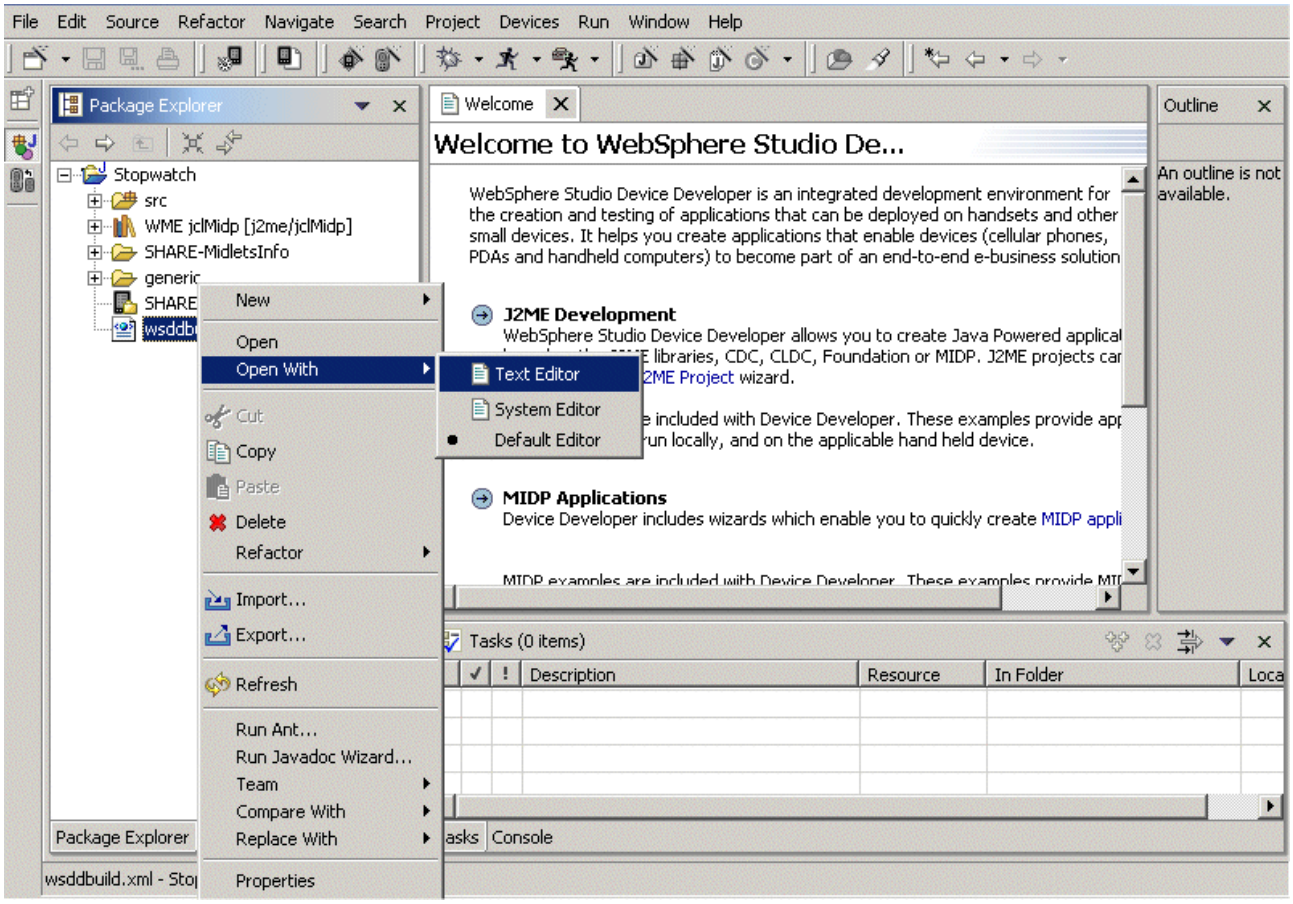
The **launch configuration** ties the **build** and **device** together. A project can have multiple build and multiple launches. Devices are shared among all projects, so any project's launches can use it.
The basic platforms are **PalmOS**, **PocketPC**, **Windows x86**, **Linux x86**, and **generic JAR**. Other device targets can be downloaded from the Device Developer's **Update Manager**.
Note that a generic **JAR** file is standard Java, and can be executed on any device. The device specific builds will create **JXE** files (more on this later), which are pre-compiled and target a specific processor environment.

Java applications that conform to Java 2 Standard Edition (J2SE) can also be built in Device Developer. These are typically run on desktops, rather than small devices, due to their resource requirements but provide a richer API. The building of a J2SE application is automatically managed by Device Developer and needs no manual intervention. This will be demonstrated in the next section.

Configuration files

Each project contains a **wsddbuild.xml** file which specifies what to build and how. Adding a build will create a hidden **bin** directory to contain output files, and modifies the XML file. Although the bin directory does not appear in the Package Explorer view, the compiled classes are present on the file system (within the Stopwatch workspace) and can be viewed using a normal file manager program. Performing a build actually creates the output files.

- View the **wsddbuild.xml** file in the Stopwatch project by clicking the right mouse button whilst the cursor is over **wsddbuild.xml**. A popup menu will appear. From this, select **Open With** -> **Text Editor**.



The contents of the file will appear and should contain the following lines.

```
<?xml version="1.0" encoding="UTF-8"?>
<builds>
    <build build-id="com.ibm.ive.jxe.builder.GenericJarBuildManager"
        folder="generic" name="Generic Build"/>
</builds>
```

This file lists each build in the project. There is only one in this case – it is a Generic JAR build and is contained in the **generic** folder. Expanding this folder reveals another XML file called **build.xml**. This is actually an Ant script – the Ant processor is used to build the code based on the entries in this file. You can add as many builds as you need, and then build them as necessary. You may also view (and edit) the Ant source, if necessary.
There are some other files in this folder and all based on a specific build. Each build creates either a **JAR** or a **JXE** file for the target device. In the case of Stopwatch, it is a JAR file, which was build without a specific target device in mind.

Launching an application

A launch links a particular build to a specific device. You need to verify that the build you choose will actually function on the device. If you get an error or the Save button will not activate, make sure the correct build has been selected. More details are provided in the “MicroAnalyzer” section.

MicroAnalyzer

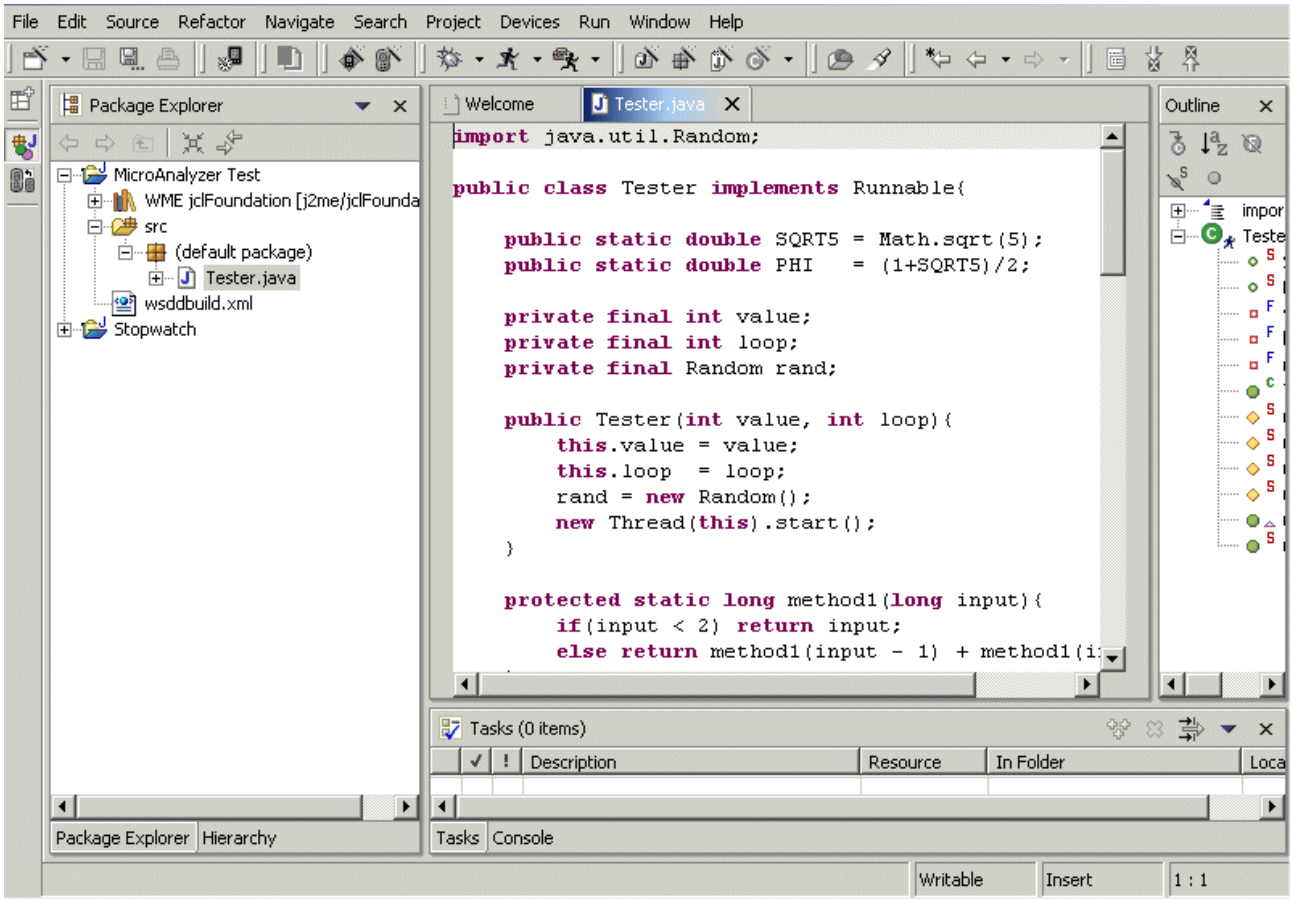
Embedded devices generally have very limited resources and are certainly not as powerful as desktop computers. For this reason, it is essential that applications are developed as efficiently as possible. The MicroAnalyzer is used to profile and analyse the execution of applications on an embedded device.
The MicroAnalyzer works alongside the built-in debugger; debugging tools can help to find and correct errors in the code, the analysis tools are then used to determine which parts of the code needs tuning and which sections are most heavily used. So, debug to get it running and then analyse to get it running well.

Both debugging and analysis can be performed locally (inside the emulator for MIDlets) and remotely (on the physical device). For remote access, the device will need an IP address, which means it needs to run TCP/IP.
A standard view for debugging applications is provided whether they are running inside the local JVM or on a remote device and since Device Developer is based on Eclipse, your debugging skills are transferable. This ability to debug any Java application on any of a variety of devices using the same tools and user interfaces has a positive effect on productivity -- forget proprietary tools provided by manufacturers and inconsistent development environments for each device.

This lab assumes experience with the debugger and so focuses on the MicroAnalyzer. For this, we will use another application that calculates the nth number in a Fibonacci sequence (starting at 0) using one of four methods. This program uses multiple threads to perform this calculation multiple times, concurrently. We will use the MicroAnalyzer to learn about the threads, and determine which method is the most efficient.

- From the Device Developer Java perspective, select **File** -> **Import**.
- Select **Existing project into workspace** in the import window and then click **Next**.
- Browse down to the location where the project is located. The instructor will provide details on where the Stopwatch lab is located. Click **Finish**.

The imported project, called **MicroAnalyzer Test**, will appear in the Java perspective with the Stopwatch MIDlet. Expand the **MicroAnalyzer Test**, **src** and **(default package)** folders to reveal the **Tester.java** file. Double-click this to open in the Java editor.

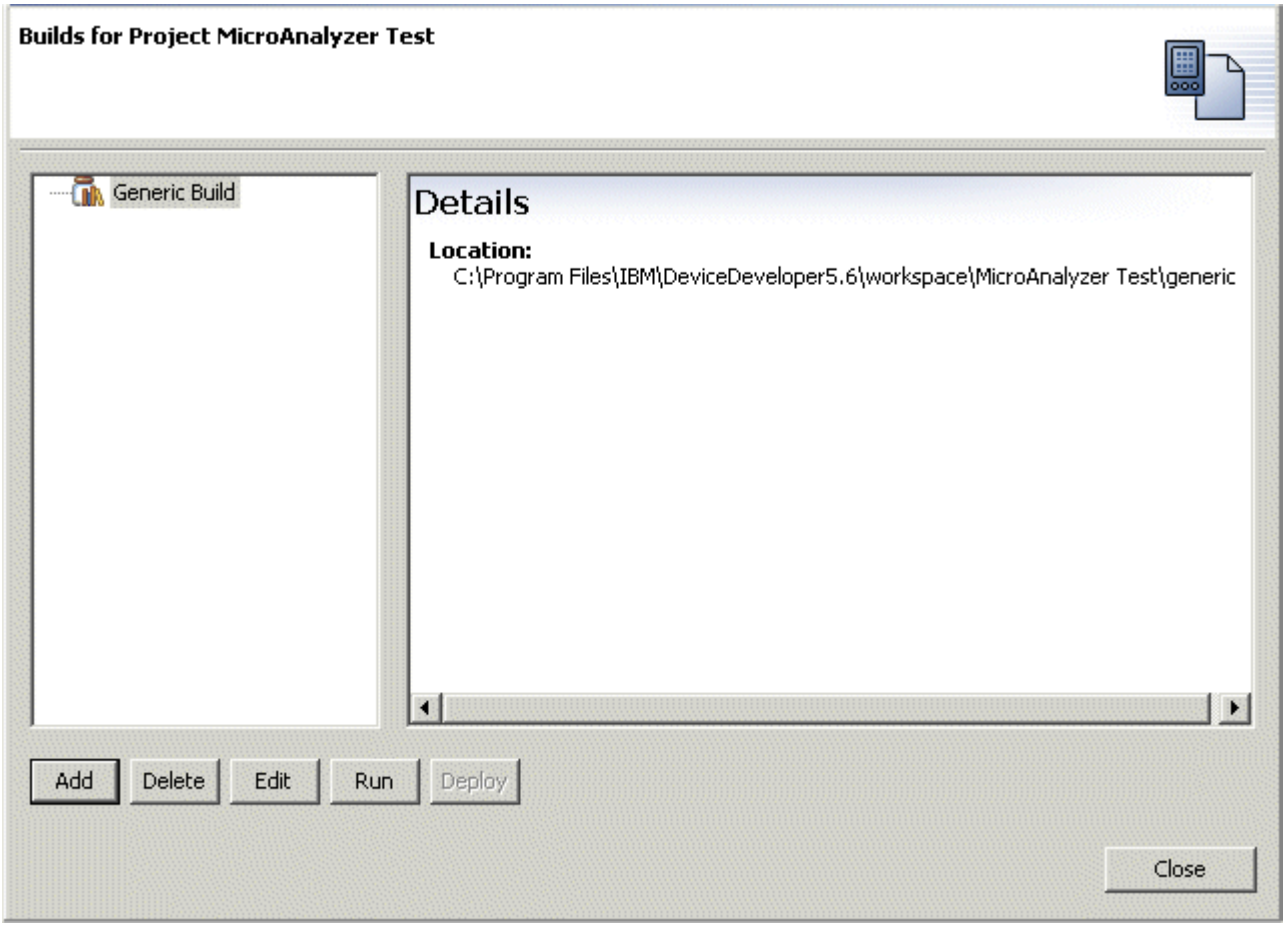


You may notice that this class does not contain the familiar MIDP lifecycle methods. In fact this application has not been designed to run in a MIDP environment. As mentioned earlier, programmers can also write Java applications that use the J2SE APIs.

- From the menu bar, select **Run** -> **Run As** -> **Java Application**. This will display the output of the program in the **Console** view. You should see a list of multiple threads, in multiple loops, using multiple methods. If you cannot easily see the contents of the Console view, the window can be resized using the mouse.

Now that you have verified the application runs in the default JVM, a build and launch configuration can be created.

- Select the **MicroAnalyzer Test** project in the Package Explorer view. Select **Project** -> **Device Developer Builds** from the menu.
- The **Configure builds** window will appear. Click the **Add** button.
- Select **Generic JAR** and click **Next**.
- The application name can be left as **Tester**. Ensure that the option to **remove unused** classes is checked. Obfuscating code can help with compaction but can be left unchecked. Click **Finish**.

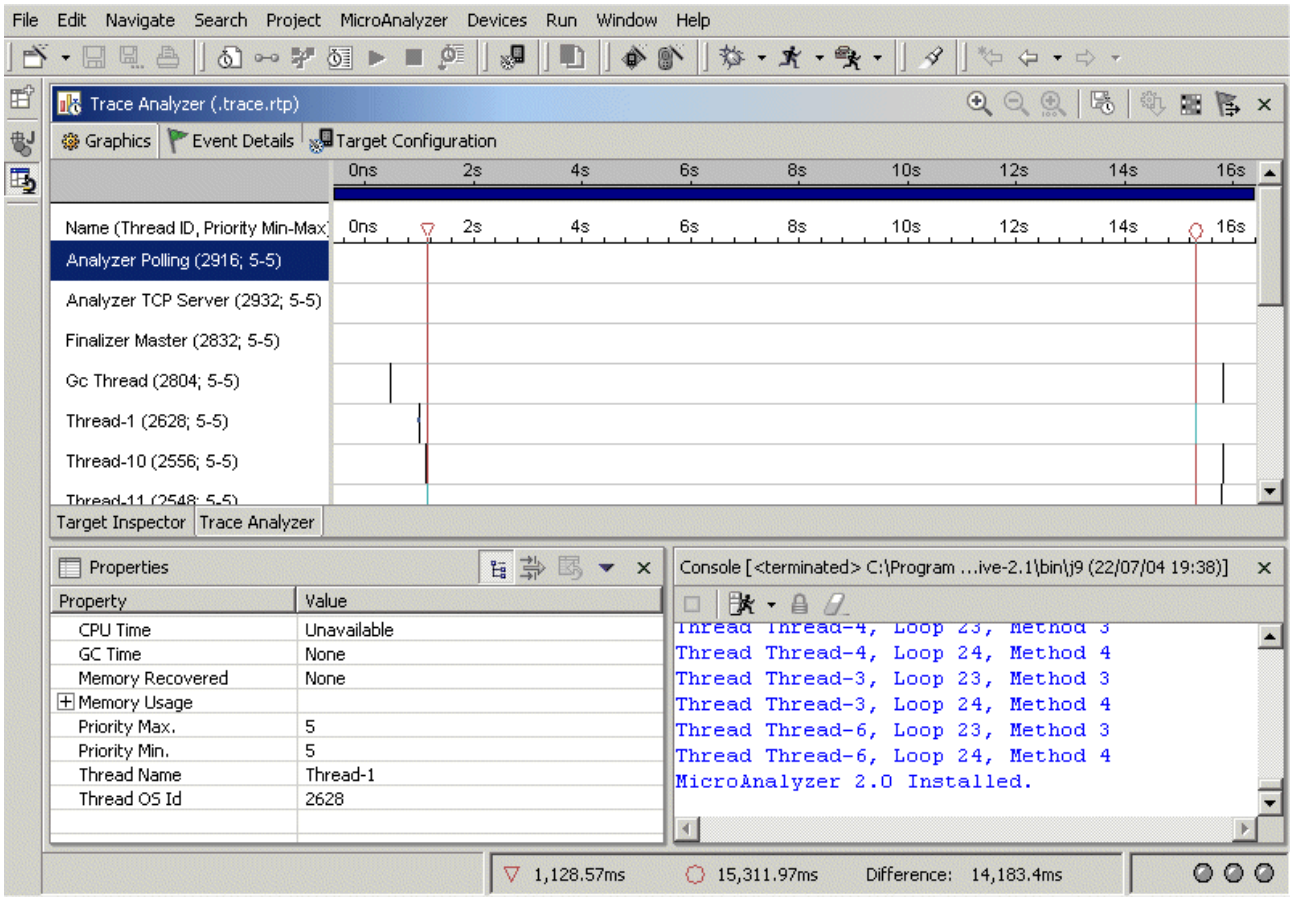


- From the **Configure builds** window, click **Close** to reveal that a build called **generic** has been added in the Package Explorer. This folder contains build information for generic Java builds that can be used on any platform without modification.
- Select **Run** -> **Run…** from the menu bar.
- A new launch configuration is created here. Select **Java on Device** and click **New**.
- Enter “MicroAnalyzer” in the **Name** text field (without the quotes).
- Click **Browse…** and select the **MicroAnalyzer Test** project. Click **OK**.
- Ensure that the **Device or JRE** is set to “ive-2.1 (local JRE)”.
- From the **Java Application** drop-down list, select “Tester.jar (Generic Build build)”. This creates a runtime configuration that associates the test application with the generic build we have just added.
- Click **Apply**.
- Select **MicroAnalyzer Configuration** and click **New**.
- Select **Java on Device : MicroAnalyzer** from the **Launch configuration to be analyzed** drop-down list. This prepares Device Developer to launch the new MicroAnalyzer configuration.
- Check the **Wait for target to connect** option. Then select **Start tracing (use Stop Trace action or wait until application terminates)**.
- Click **Apply** then click **Run**. There will be a delay whilst Device Developer builds the class files for the generic build (it is necessary to build class files for each defined build, as we will see later).

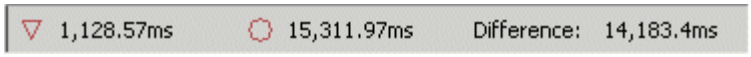
Wait until the binaries have been built for the new build. The MicroAnalyzer will connect to the running application, analyse its behaviour and then once the application has finished, it will disconnect. During analysis, the console should display the output from the running application. You should receive a message stating that the connection to the target program has terminated. At this point, analysis has completed and can be viewed.

- Click **OK** to accept the termination message.

You should now be presented with the **Trace Analyzer**. It is possible to use this view to determine how much time each part of the application is taking.



The start and stop events for each thread are visible as vertical lines. Move the cursor over a line for more information. To determine the time between two events, move the event markers (red lines with a triangle and a circle above – you must drag the actual triangle or circle). The time difference will be displayed at the bottom of the perspective.



The zoom icons  can be used to magnify an area for closer inspection. When zooming in, the blue bar at the top of the display shrinks. This bar represents the visible area of the display. You can click and drag the bar to display different parts of the zoomed display.

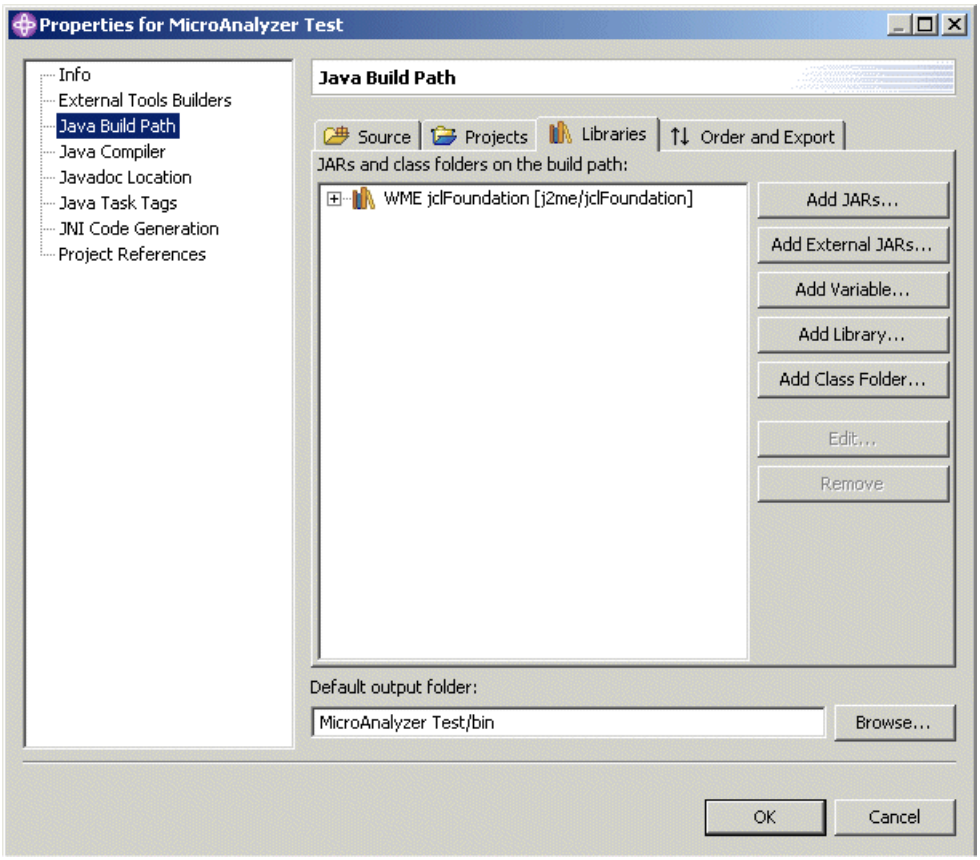
- Have a look at the threads and see when they started. You should find that the majority were started by the JVM at roughly the same time (with the exception of the GC and the main thread). However, the times they finished will differ.
- To get a different view of thread timings, click the **Event Details** button. This shows the same information in a table. Expand each thread to reveal the timings and memory usage.

Every colour in the Trace Analyser has a meaning. To check what each colour represents, click the **Colors…**  button.

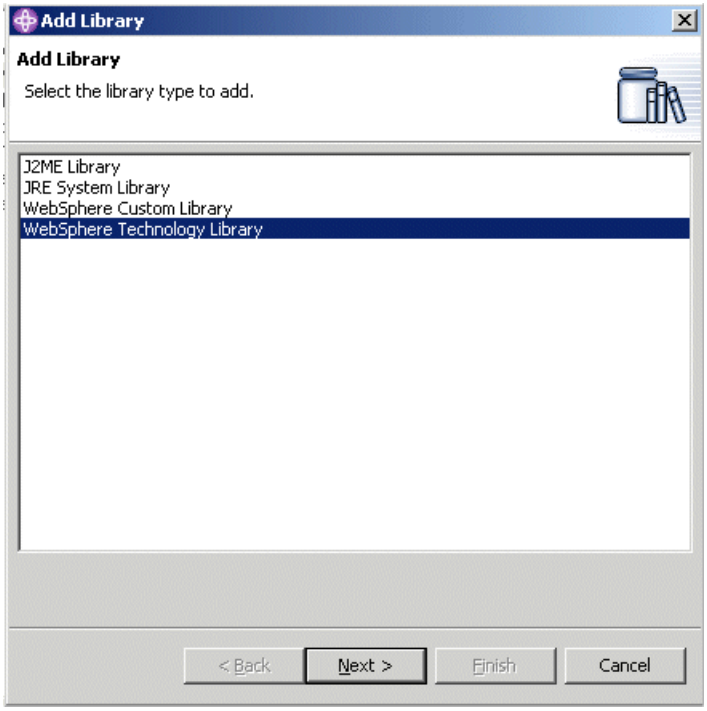
User events

MicroAnalyzer provides a Java Native Interface (JNI) call to generate user events. These simply mark points in the code's execution. Typically, they are used to bracket interesting or suspect parts of the program code so that the trace displays the entry and exit points. For example, user events can be inserted before and after a loop. The time spent in the loop can be determined by comparing the times of the two user events in the trace display.
User events are identified by numbers between 100 and 65535 (all numbers under 100 are reserved for use by the MicroAnalyzer). To add a user event to the code, a class needs to be imported which is provided by the **WebSphere Technology Library**.

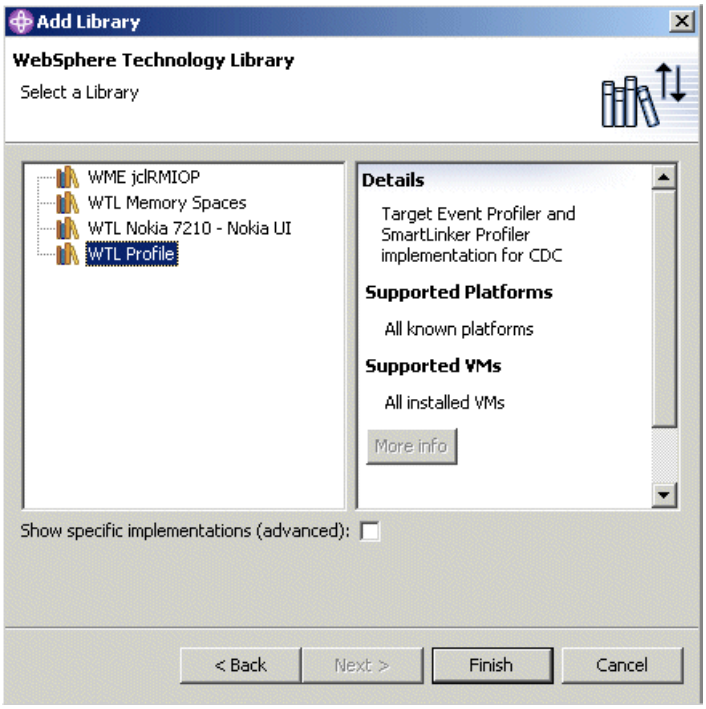
- Switch back to the Java perspective, right-click the **MicroAnalyzer Test** project and select **Properties**. Select **Java Build Path**.



- We must include the **WebSphere Technology Library** on the build path before we can import the required class for debugging user events. Ensure that the **Libraries** tab is selected (as shown in the figure above) and click **Add Library…**



- Select **WebSphere Technology Library** and click **Next**.
- Select **WTL Profile** from the list (which contains the required classes for debugging user events) and click **Finish**.



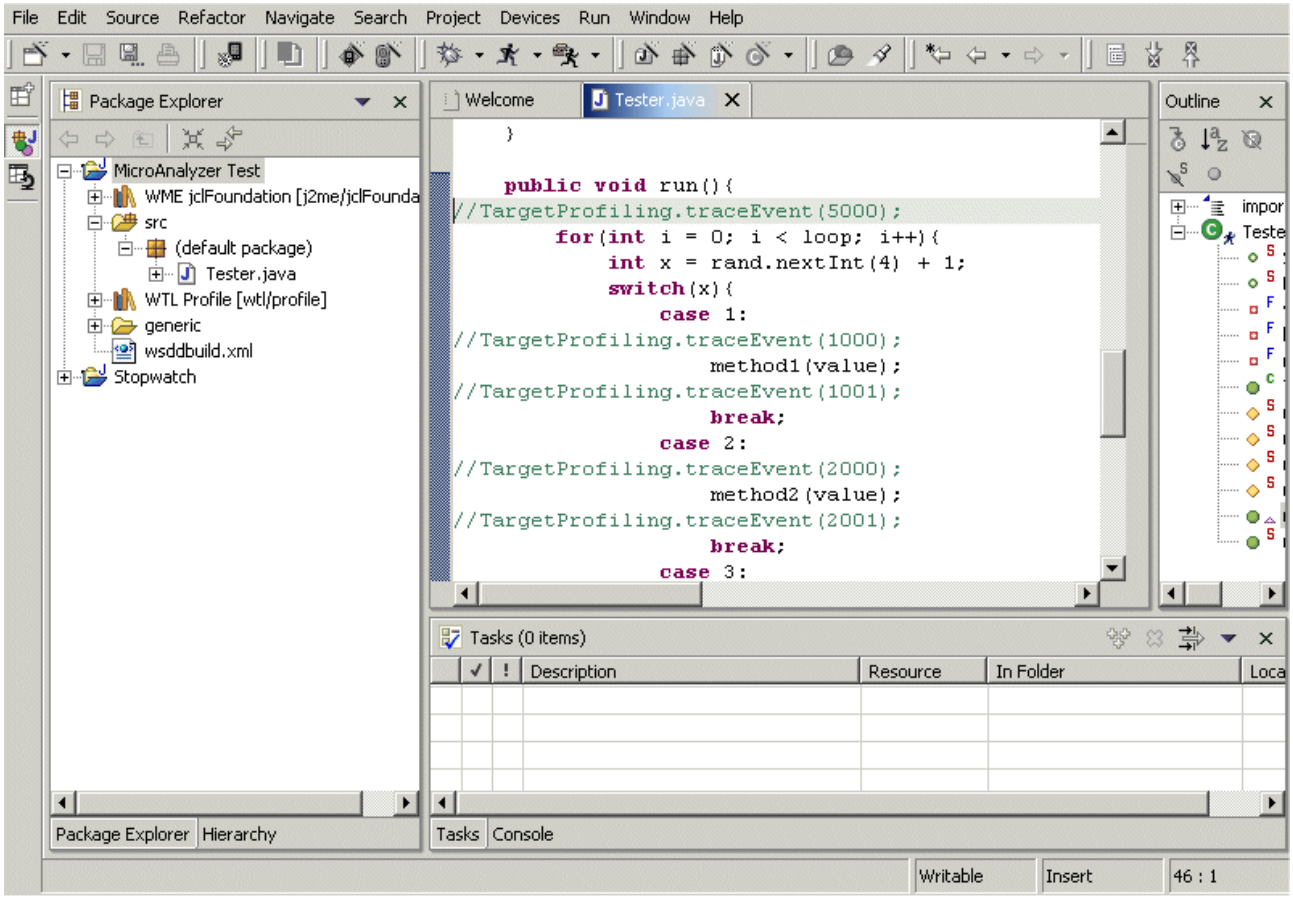
- The new profile will be added to the build list. Now click **OK** to close the Properties window.
- Ensure that the **Tester,java** file is open in the Java perspective (if not, then double-click **Tester.java** to open it) and scroll down to the **run** method.

You should see some lines that are commented out. Each line contains the following code followed by a number (in parentheses):

```
TargetProfiling.traceEvent
```

Each line defines a user event and associates a unique number. The number allows each event to be easily tracked during debugging and can be inserted anywhere in the code. In this sample code, the

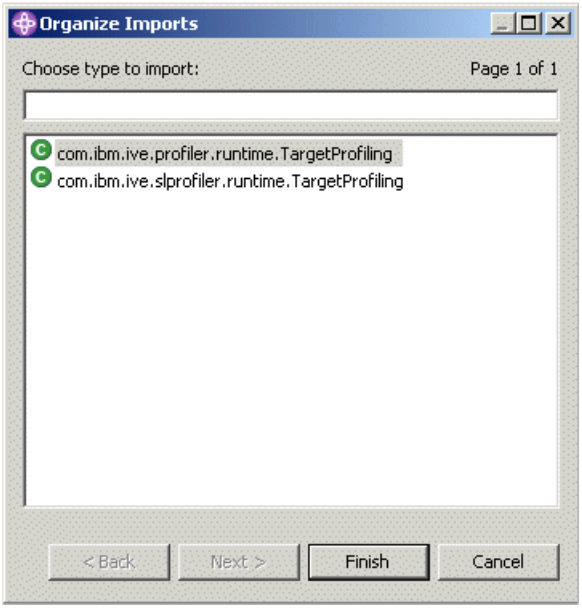
following numbering system has been used – the events have been paired, where the “start” event ID number is a multiple of 1000 and the “end” event adds one to that number.



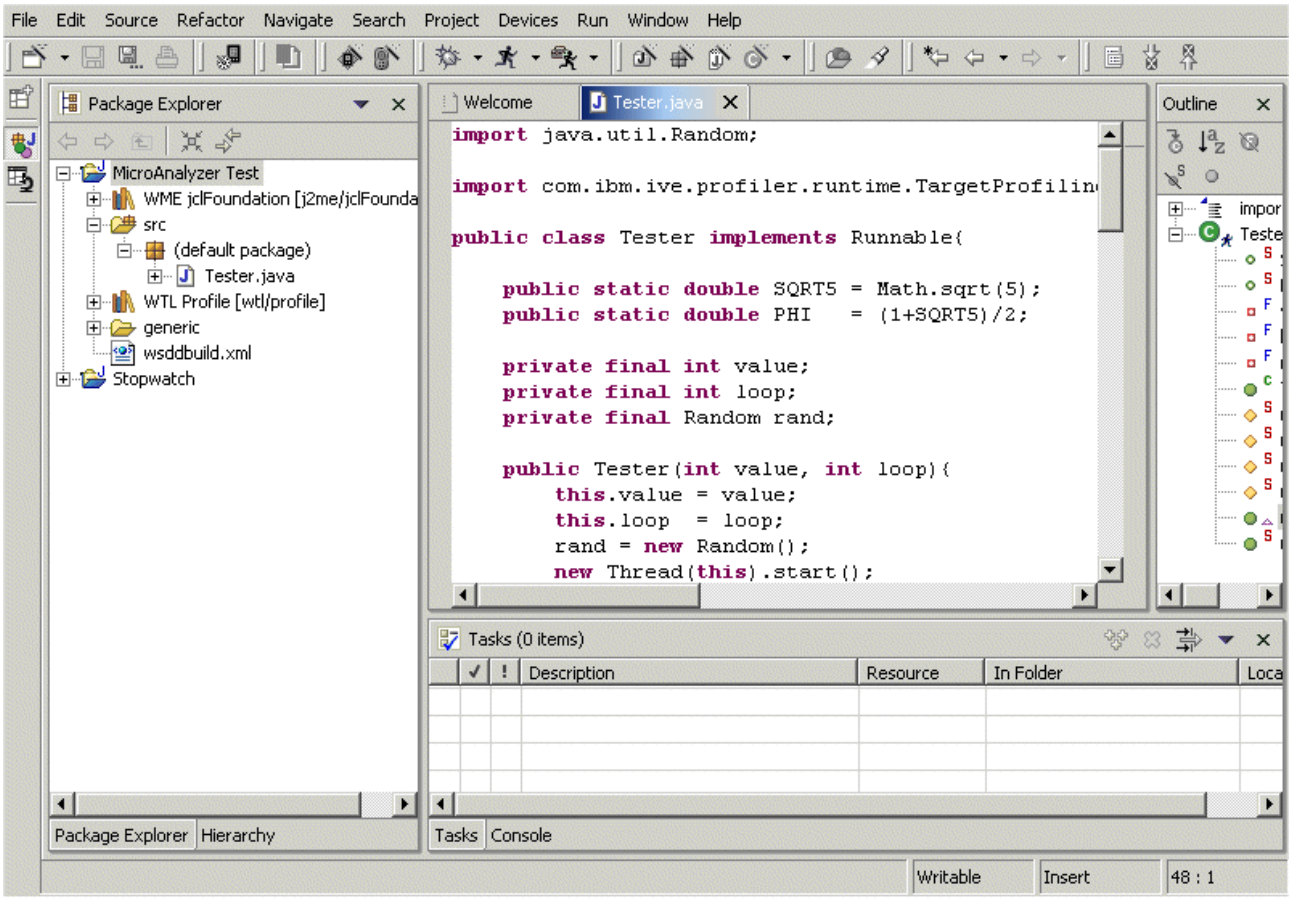
- Uncomment each line in the **run** method and save the file. A commented line starts with a double-slash //. Simply remove the double-slash to uncomment (the colour of the line will change to indicate this it has been uncommented).

A list of compilation errors will appear – this is because the compiler cannot resolve the profiling class. We must add an **import** statement to help the compiler resolve this. This could be achieved manually by simply locating the appropriate class and adding the code in the import section. However, we will try out a nice facility in Device Developer to help us find the class and import it automatically.

- Right-click inside the source editor view and select **Source** -> **Organize Imports**.

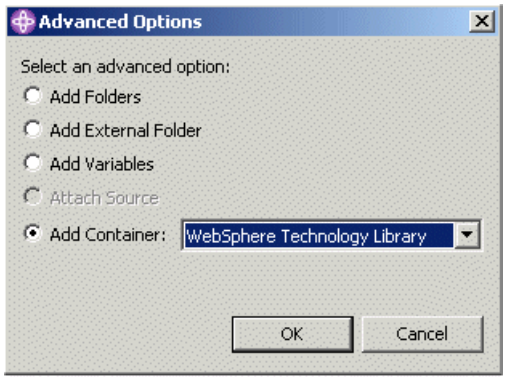


- All classes that are found on the Java build path that match **TargetProfiling** will appear. Select **com.ibm.ive.profiler.runtime.TargetProfiling** and click **Finish**. This will add an import statement in the code (the import section is near the beginning – scroll up to the top to confirm this).
- Save the file again and ensure that there are no more compilation errors (by checking that the Tasks list is empty).

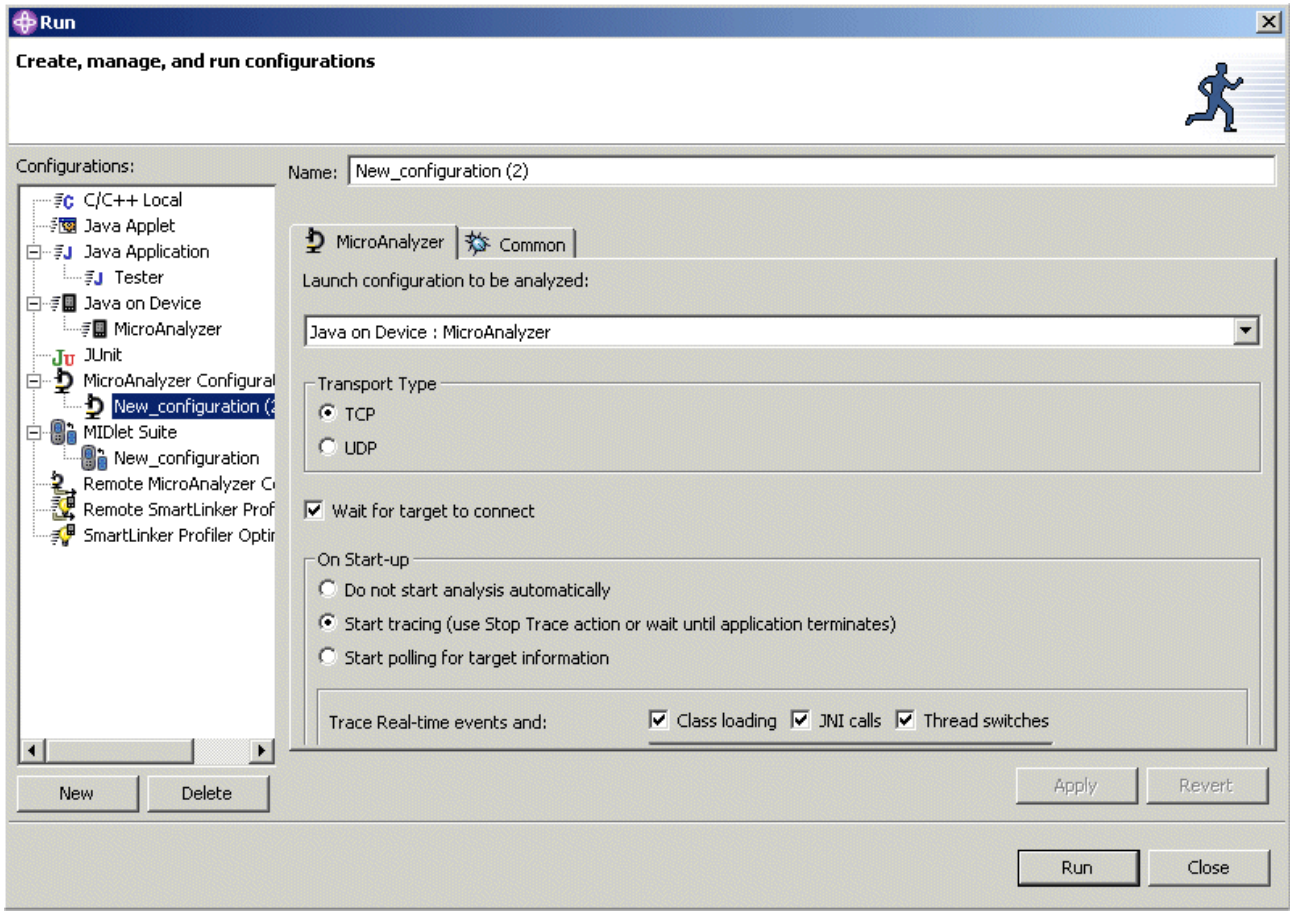


- Select **Run** -> **Run…** from the menu bar.
- Select **Java on Device** -> **MicroAnalyzer** and click the **Classpath** tab. You will now configure the Java runtime classpath to include the **WebSphere Technology Library** (we only set the Java build classpath previously).

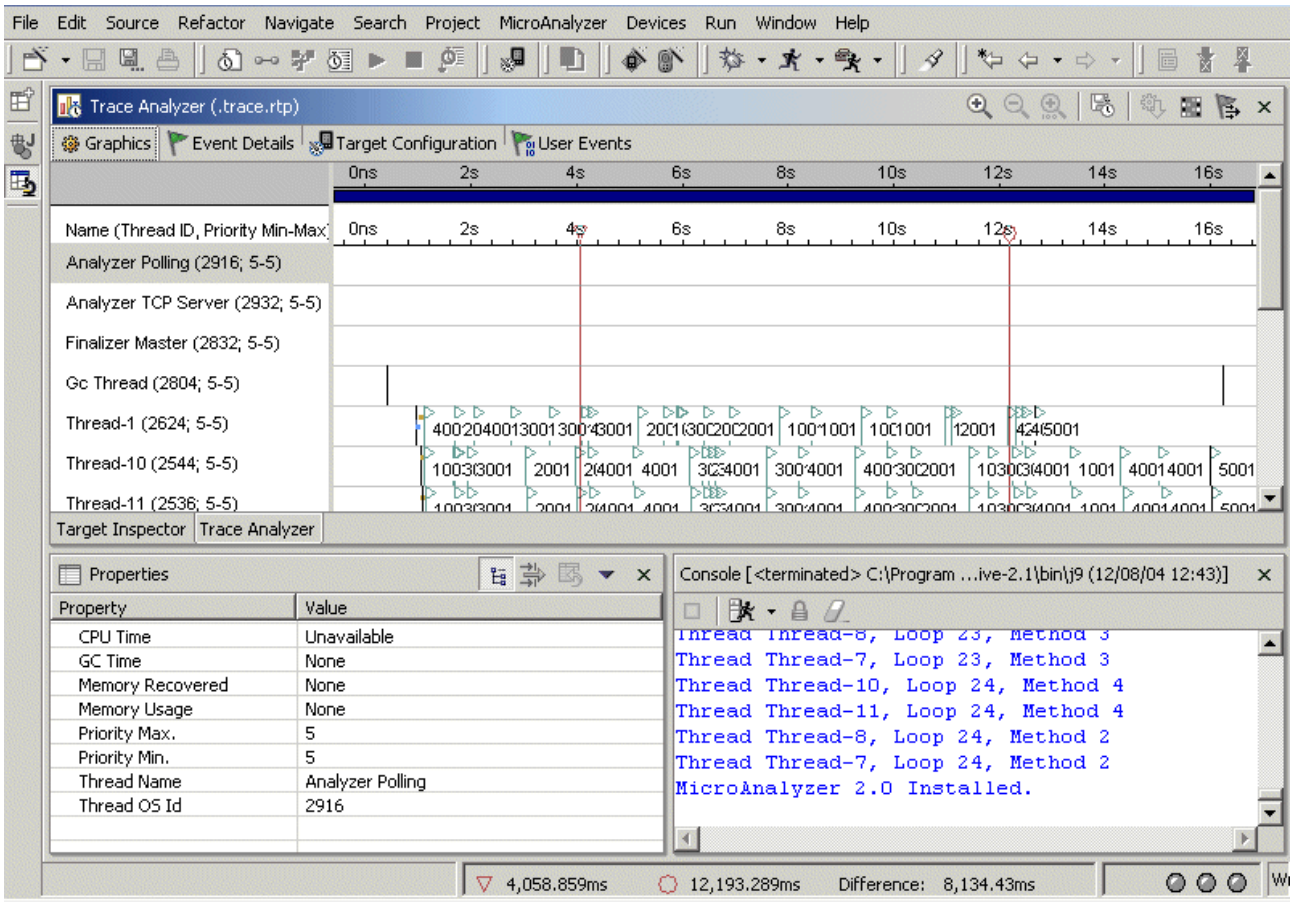
- Uncheck the **Use default class path** check box.
- Click the **Bootstrap classes** tab and click **Advanced…**
- Select **Add Container** and choose **WebSphere Technology Library** from the drop-down list.



- Click **OK**.
- Select **WTL Profile** and click **Finish**. (This process should be familiar but the Java compiler and Java runtime use different classpaths, both of which must include the WebSphere Technology Library!)
- Click **Apply** then select **MicroAnalyzer Configuration** -> **New_configuration (1)**. This will return to the MicroAnalyzer launch configuration that was created earlier to analyze our application.

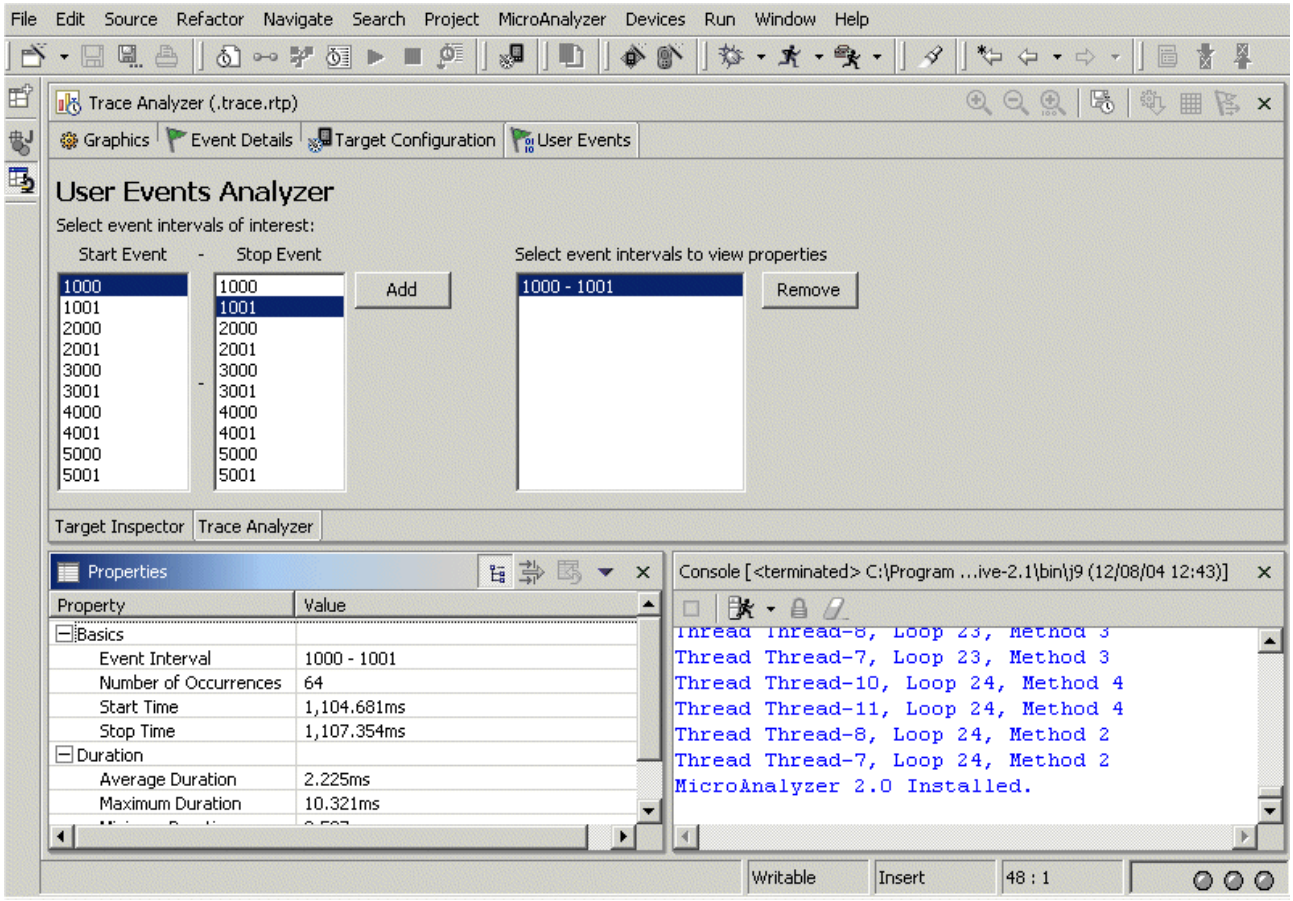


- Click **Run** and wait for the application to complete.
- You should receive a familiar message informing that the target program has terminated. Click **OK**.
- The MicroAnalyzer perspective should re-appear displaying the trace with the user events included.



Each user event is displayed with a blue flag and a number (which is the ID number set in the source code). If the numbers do not appear, then they can be enabled by clicking the **Trace Analyzer View Preferences** icon and checking the **Show User Event numbers** box.

- Click the **User Events** tab User Events for a breakdown of start and stop events. This view provides details of timings between user events.
- Select **1000** in the **Start Event** column and select **1001** in the **Stop Event** column. Click **Add** to add this selection to the third column. Detailed information regarding the time between these two events will appear in the **Properties** view.



Other intervals can be accurately monitored using this method. This tool provides a convenient means of optimising code for performance to ensure the best application response time.

Device Developer also supports tracing an application running on an actual device. The application can be traced running remotely and the same view of this data is presented.

SmartLinker

One of the biggest challenges for mobile device developers is how to make the most efficient use of limited device resources. The SmartLinker helps Java applications fit into and perform better on resource-constrained devices.

Pre-linking

SmartLinker prelinks all of a Java application's classes into a single data structure written out as the ROM.class file. This file, along with any supporting resource files and metadata, is packaged for deployment in a JXE, or J9 executable file, which can be executed by the J9 VM on the device. JXEs are executable JAR files built by the SmartLinker. They are specific to the Micro Environment (see “WebSphere Everyplace Micro Environment” on page 4 for details) and the Custom Environment (see “WebSphere Everyplace Custom Environment” on page 4). If you need to deploy to other JVMs, do not select this option. When specifying the build, you have a choice of JXE or JAR output.
JXE files are platform-specific, which means you will have to build a different JXE file for each launch platform you choose to support, however, since they are pre-compiled, they will provide better performance (and often load faster) than a JAR file.
JAR files are traditional Java deployment packages – code in a defined format, zipped into a file. These files may be used on any platform.

Code reduction

The code reduction capabilities of SmartLinker let you conserve runtime memory by paring down your application's subsequent binary to its bare minimum.
The extent of the reduction is up to you. SmartLinker analyzes your program and identifies all types, methods and/or fields that are referenced during execution on a specified target device. Then, it eliminates everything that is unnecessary, leaving only what is needed to execute your program. Furthermore, SmartLinker's code relocation feature allows you to save even more RAM by letting you put your application in ROM or flash memory. If you wish, you can specify the ROM flash address where the code is to reside.

Ahead-Of-Time (AOT) compilation

An often-used approach to speed up the execution of Java applications is to use Just-In-Time (JIT) compiler technology. In cooperation with the J9 VM, the JIT compiler instantaneously translates bytecodes into native machine code.

The main advantage of JIT is that you can achieve a significant performance improvement without losing the platform-independent nature of the bytecode. The main drawbacks are that the JIT compiler takes up a lot of space and consumes CPU cycles every time you run your application.

As an alternative, SmartLinker allows you to translate bytecodes into native machine code at link time, or ahead of time (AOT). When SmartLinker packages a set of classes in a JXE file, it generates native code for the bytecode that is embedded in the JXE file. At runtime, the J9 VM directly calls the native code for all precompiled methods. Methods that are not precompiled are executed by the VM as usual.

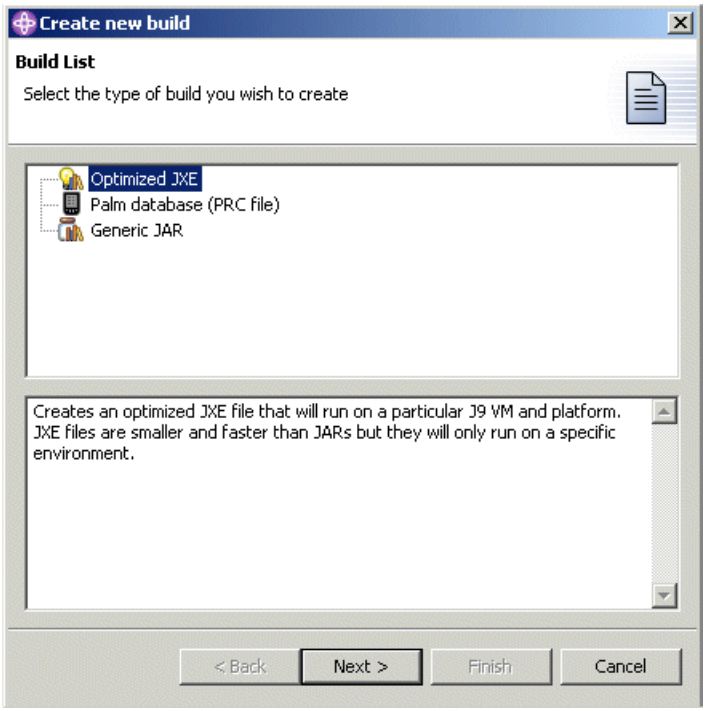
Advantages of using the SmartLinker are:

- Footprint reduction, which reduces the size of the deployed code by prelinking and optionally eliminating unused classes and methods.
- Execute in place (XIP) prelinking enables code to be run in place in ROM/Flash RAM.

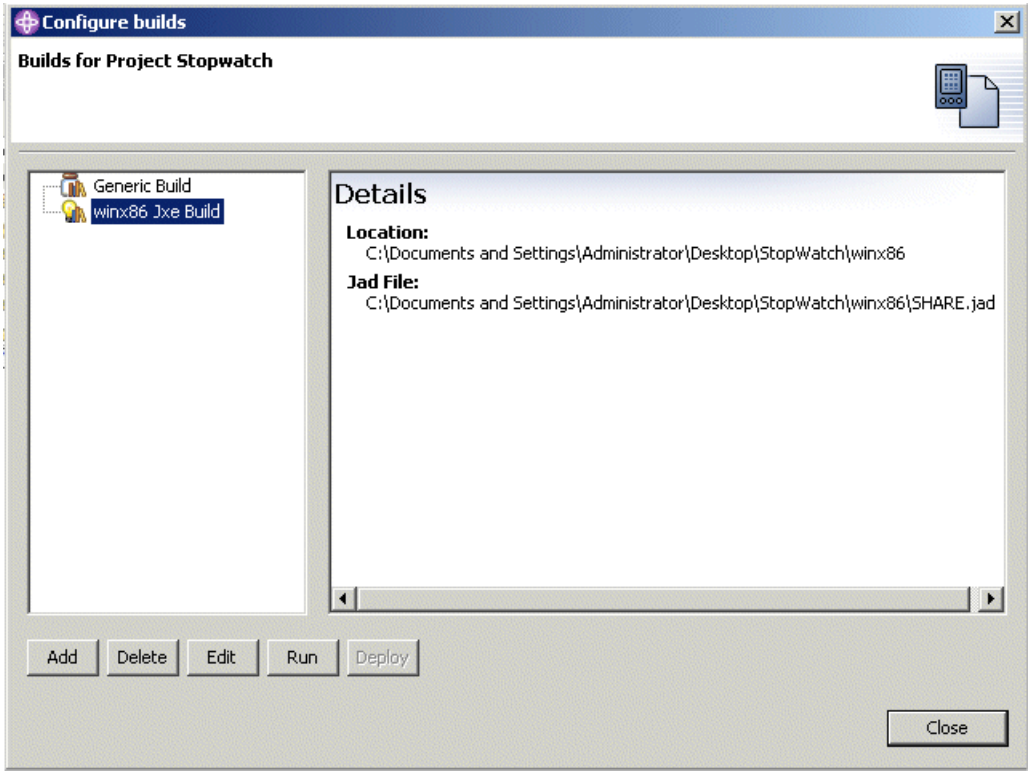
- Accelerated application startup.
- Debugging and profiling/analysis (with MicroAnalyzer).
- Dynamic loading of Java class files.
- Dynamic linking with other JXE files.
- Elimination of unused classes and/or methods.
- Segmentation, which allows JXE files to be segmented to support devices with limited segment sizes.
- Ahead-of-time (AOT) compilation (for selected targets).
- Method inlining.
- Obfuscation of names for reduction in size and security.

More information on these details can be found in the Device Developer online documentation. Now try out the SmartLinker by returning to the Stopwatch project. Currently, only a generic build has been created. We can easily create optimised builds from the same source code.

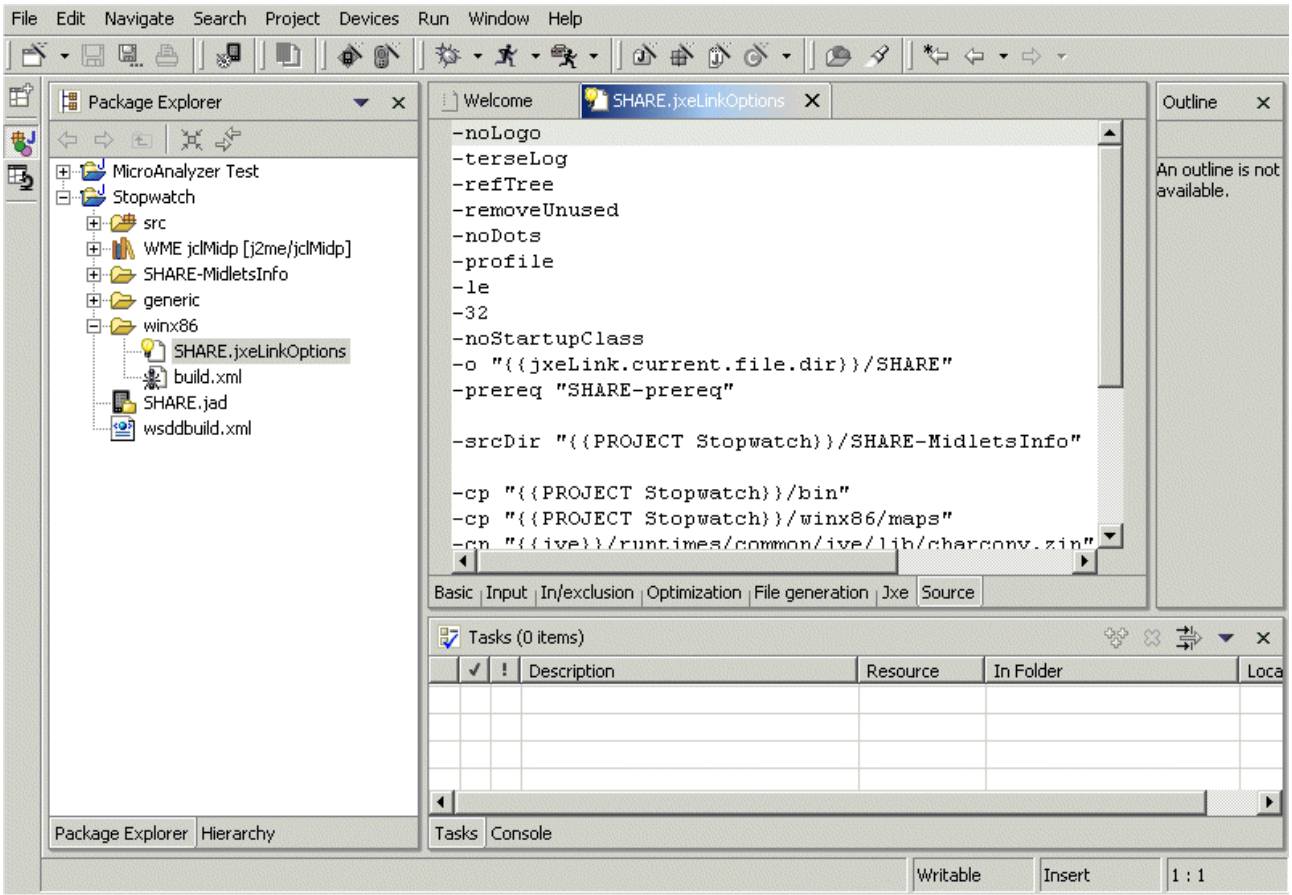
- In the Java perspective, right-click on the **Stopwatch** project and select **Device Developer Builds**.
- Click **Add**.



- We have the option to create an optimised JXE file. Ensure that this option is selected and click **Next**.
- Ensure that **Remove unused classes** is checked and **Obfuscate code** is not checked. Obfuscating can further optimise the code size but will not be demonstrated here. Click **Next**.
- Select **J9 for Windows x86** from the **Platform** drop-down list.
- Click **Finish**.
- The new build will be added to the build list. Click **Close**.



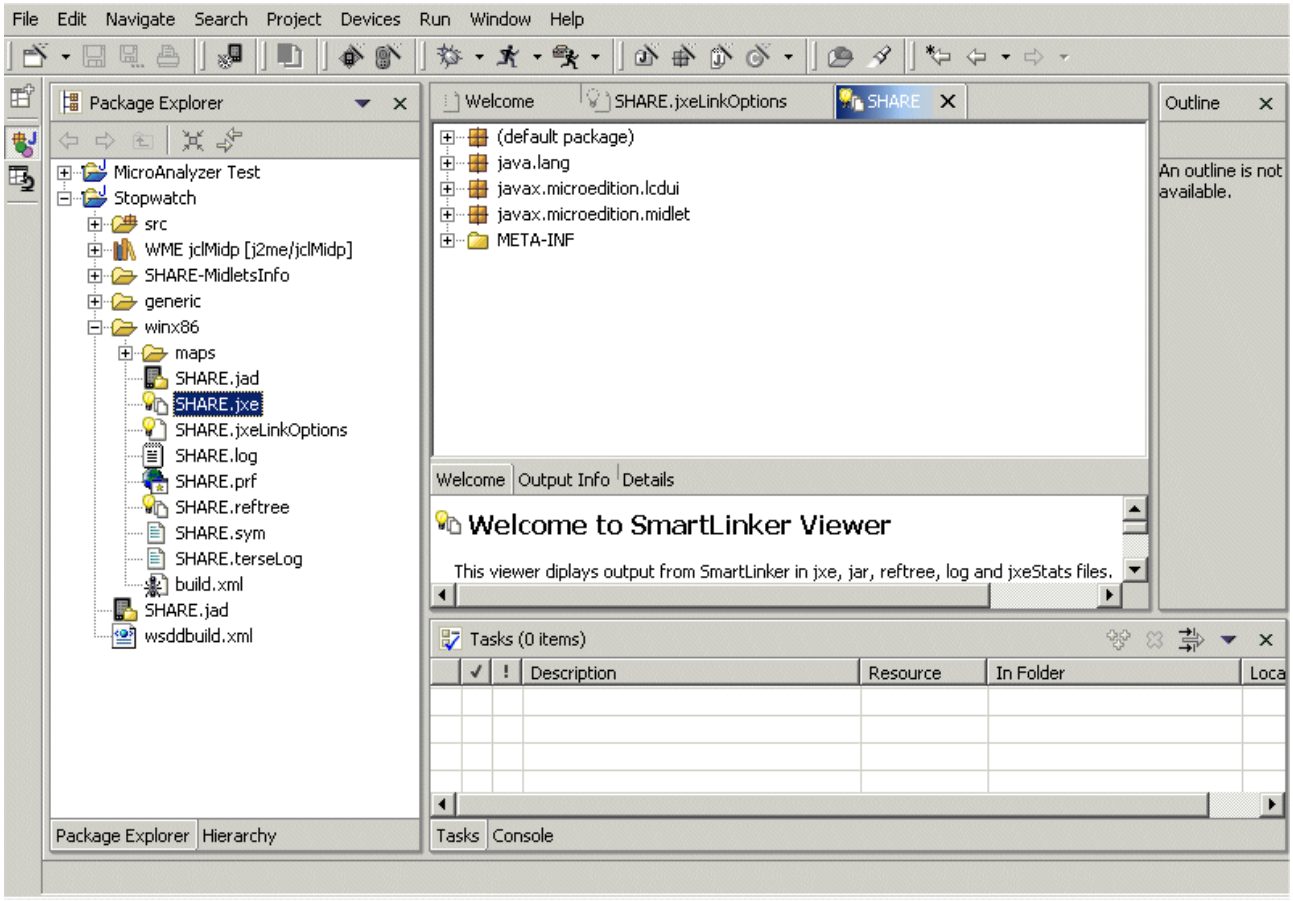
- A new build folder called **winx86** has been added to the project. Expand this in the **Package Explorer** and open the **SHARE.jxeLinkOptions** file.
- Click the **Source** tab for the newly-opened file.



Each target platform (in this case **generic** and **winx86**) will contain a **jxeLinkOptions** file for that platform (as shown above). This file contains all the SmartLinker options that are used to build JXE files for that specific target platform.

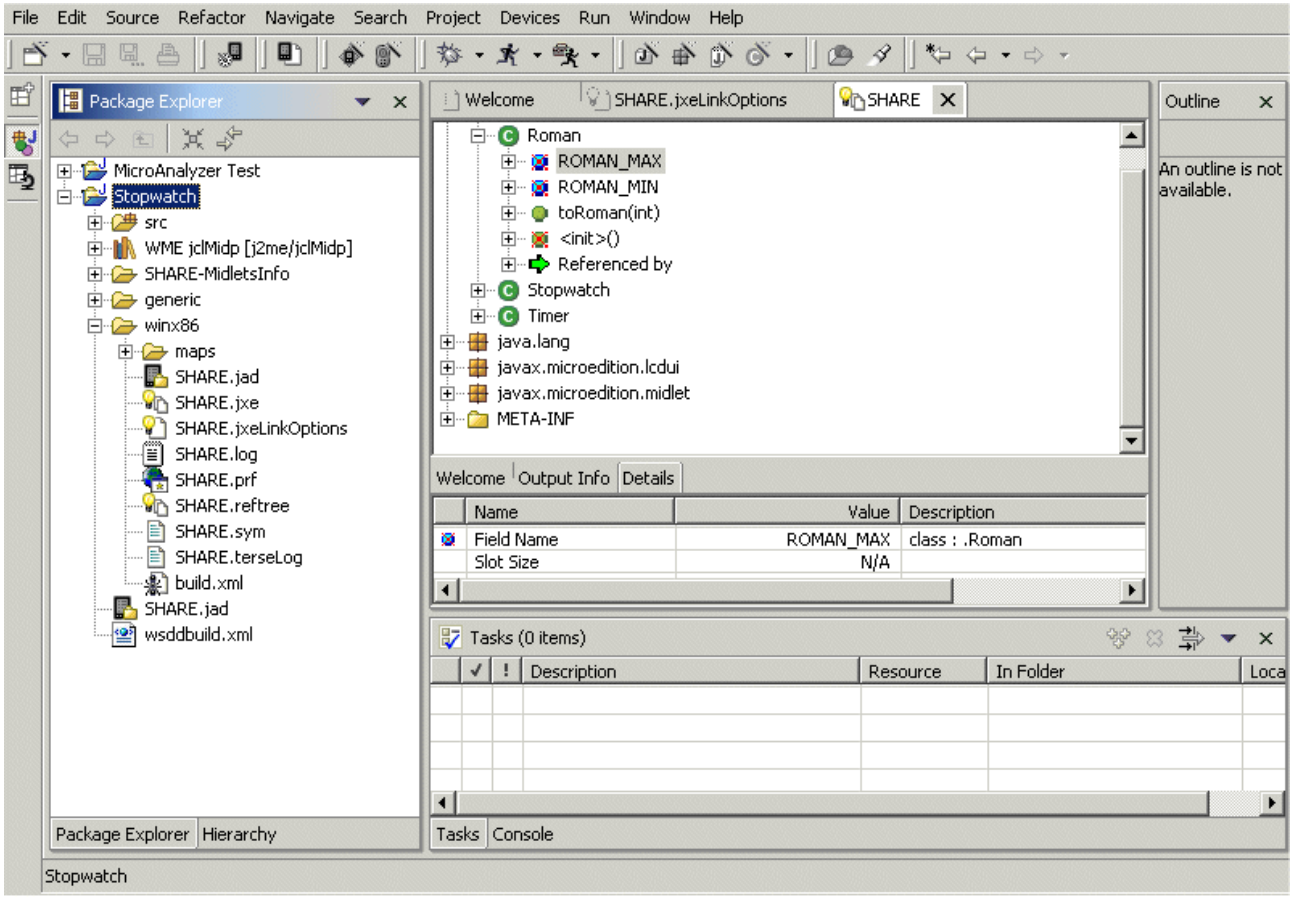
The settings can be edited either manually on the Source tab or by using the various other tabs in the editor. As can be seen, there are many options that can be tweaked and fortunately the defaults will perform some optimisations. Refer to the Device Developer documentation for details of each option.

- To build a JXE for winx86, right-click the Stopwatch project and select **Device Developer Builds**.
- Select **winx86 Jxe Build** and click **Run**. Building for a target may take several seconds. During this time, a log is created.
- Click **Close** when the build is complete and then click **Close** a second time.
- The SHARE.jxe file should be visible in the winx86 folder. Double-click to open.



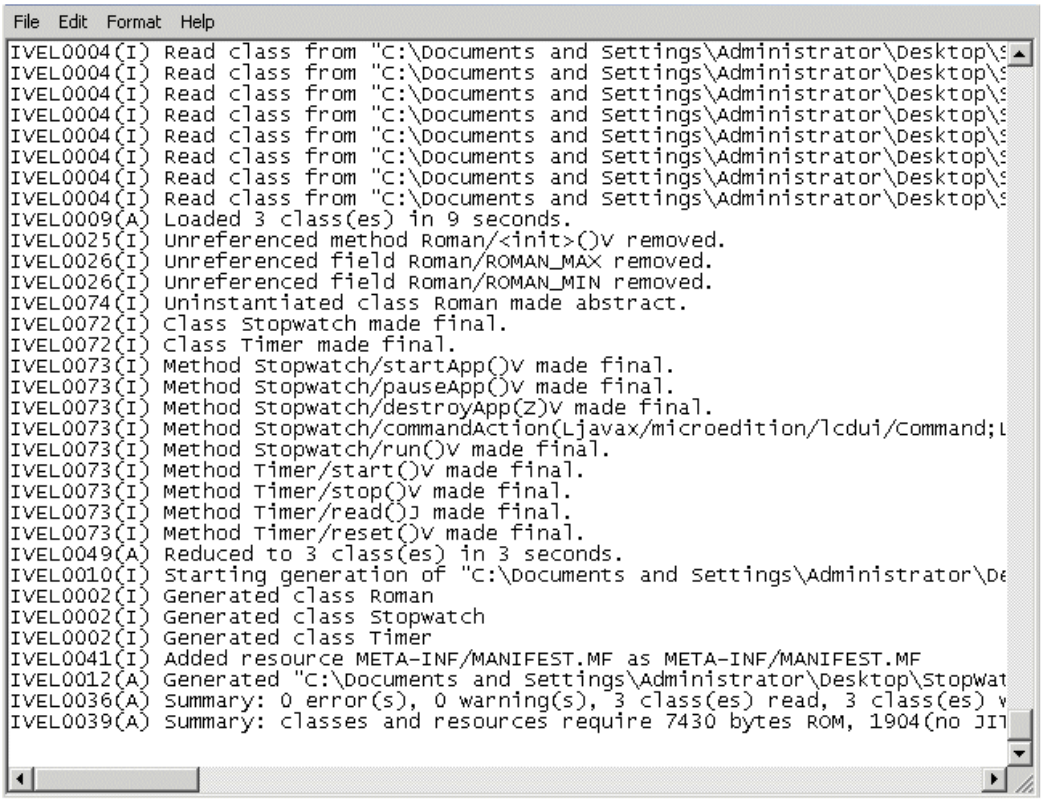
Each package that has been compiled into the JXE is displayed. However, only the classes that are referenced have been included.

- Expand **(default package)** to reveal the application classes.
- Expand the **Roman** class. Each method and field in the class is listed, however, those that have been optimised out of the build have a red cross through them. In this case, the **ROMAN_MAX** and **ROMAN_MIN** fields (defined as static) and the class default constructor (never referenced) have been removed.



Clicking the **Output Info** tab will display more details about the JXE, including how much ROM and RAM will be required to deploy the JXE.

- Double-click **SHARE.log** (in the Package Explorer view) to open the log generated during the build of this JXE file. Scroll to the bottom to find what optimisations have been made to the application.



This shows that a number of methods and fields have been removed and that several methods have been made final (again for optimisation purposes).

Deploying on a device

If you have a mobile device that you'd like to deploy the Stopwatch application on, it must be MIDP-compliant. You will need to ensure that a JVM is installed on the device, which can often be downloaded from the manufacturer's Web site. IBM provides optimised JVM's for several types of device, including Pocket PC, Palm and Linux.

The steps for deploying on an actual device are similar to deploying in an emulated environment as demonstrated in this lab. A device configuration (which describes how to communicate with the device) and a Launch configuration (which associates a project, build and device) are required for the appropriate device. The Device Developer online help is the best place to start. Select **Help** -> **Help Contents** from the menu to access the contents page.

Further reading

"Using WebSphere Studio Device Developer to Build Embedded Java Applications", (http://www.redbooks.ibm.com/redbooks/pdfs/sg247082.pdf).

WebSphere Client Technology, Micro Edition (WCTME) is the latest IBM platform for the evolution of pervasive computing. This platform provides a family of products for deploying "high-value data services on mobile devices to enable and extend the on-demand enterprise". For more information, refer to http://www-306.ibm.com/software/wireless/wme/wctme.html.