```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
public class CalculatorEngine implements ActionListener {

   public void actionPerformed(ActionEvent e){
       // An empty method body
   }
}
```

Even though the `actionPerformed()` method doesn't contain any code yet, it's considered implemented in **Listing 9-2** (the curly braces make the compiler happy). JVM calls this method on the class that implements the `ActionListener` interface whenever the user clicks the button.

The next version of `CalulatorEngine` (see **Listing 9-3**) will display a message box from the method `actionPerformed()`. You can display any messages using the Swing class `JOptionPane` and its method `showConfirmDialog()`.



Listing 9-3: This class displays a message box

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JOptionPane;
public class CalculatorEngine implements ActionListener {

   public void actionPerformed(ActionEvent e){
       JOptionPane.showConfirmDialog(null,
               "Something happened...",
               "Just a test",
               JOptionPane.PLAIN_MESSAGE);
   }
}
```

Figure 9-1



If you register the class `CalculatorEngine` from **Listing 9-3** as a listener for the class `Calculator` from Listing 8-4, it'll display the message box shown in **Figure 9-1** when the user clicks inside the calculator window.

The class `JOptionPane` declares several overloaded methods named `showConfirmDialog()` — I used the version with four arguments in **Listing 9-3**. The first argument is null, which means that this message box does not have a parent window. The second argument contains the title of the message box. The third contains the message itself, and the fourth argument allows you to select a button(s) to be included in the box - `PLAIN_MESSAGE` means that it needs only the button OK.

Registering Components with ActionListener

What will call the code in the method `actionPerformed()` shown in **Listing 9-3**, and when? Register the calculator's buttons with the class `CalculatorEngine`, and JVM will obediently call this class every time any button is clicked. Add the following two lines at the end of the constructor of the class `Calculator` (Listing 8-4), and one button will start responding to clicks with the box from **Figure 9-1**.

```
CalculatorEngine calcEngine = new CalculatorEngine();
button0.addActionListener(calcEngine);
```

The other calculator buttons remain silent because they have not been registered with our action listener yet. Keep adding similar lines to bring all the buttons to life:

```
button1.addActionListener(calcEngine);
button2.addActionListener(calcEngine);
button3.addActionListener(calcEngine);
button4.addActionListener(calcEngine);
```

Finding the Source of an Event

The next step is to make our listener a little smarter: It has to display message boxes with different text, depending on which button was pressed. When an action event happens, JVM calls the method `actionPerformed(ActionEvent)` on your listener class, and this method provides valuable information about the event in the argument `ActionEvent`. In partic-

ular, the method `getSource()` in the object `ActionEvent` supplied to `actionPerformed()` in **Listing 9-3** will tell you what object was the reason for the action.

But according to Java documentation for `ActionEvent`, the method `getSource()` returns an instance of type `Object`, which is a superclass of all Java classes, including window components. Since in our calculator only buttons can possibly be the reason for an action event, cast the returned `Object` to the type of a `JButton`:

```
JButton clickedButton =  (JButton) evt.getSource();
```

Only after performing casting from `Object` to `JButton` can you call methods that `JButton` supports, for example `getText()`, which returns the button's label, as shown in **Listing 9-4**. If you press the button labeled 5, you'll see a message box that reads "You pressed 5."



Listing 9-4: Getting label of clicked button

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JOptionPane;
import javax.swing.JButton;
public class CalculatorEngine implements ActionListener {
   public void actionPerformed(ActionEvent e){
        // Get the source of this action
     JButton clickedButton=(JButton) e.getSource();
       // Get the button's label
       String clickedButtonLabel =
                             clickedButton.getText();

        // Concatenate the button's label
       // to the text of the message box
       JOptionPane.showConfirmDialog(null,
               "You pressed " + clickedButtonLabel,
               "Just a test",
               JOptionPane.PLAIN_MESSAGE);
   }
}
```

What if the window events are produced not only by buttons, but by some other components as well? Then don't cast every object that has arrived with `ActionEvent` to `JButton`. Use the operator called `instanceof` to perform

the proper casting. The next example first determines what type of object caused the event, and then performs casting to either `JButton` or `JTextField`:

```
public void actionPerformed(ActionEvent evt){

   JTextField myDisplayField=null;
   JButton clickedButton=null;

   Object eventSource = evt.getSource();

   if (eventSource instanceof JButton){
       clickedButton = (JButton) eventSource;
   }else if (eventSource instanceof JTextField){
       myDisplayField = (JTextField)eventSource;
   }
}
```

Consider the buttons that perform arithmetic operations. Our calculator has to execute different code for each button:

```
public void actionPerformed(ActionEvent e){

     Object src = e.getSource();

     if (src == buttonPlus){
        // Call the method that adds numbers here
     } else if (src == buttonMinus){
       // Call the method that substracts numbers here
     }else if (src == buttonDivide){
      // Call the method that divides numbers here
     } else if (src == buttonMultiply){
     // Call the method that multiplies numbers here

     }
}
```

How to Pass Data between Objects

When you click a numeric button on the real calculator, it does not show a message box, but rather displays the number in the text field on top. Here's a new challenge: You need to be able to reach the attribute `display-Field` from the object `Calculator` from the method `actionPerformed()` defined in `CalculatorEngine`. In other words, two objects need to communicate. There are different ways of arranging this: For instance, in the class `CalculatorEngine` you can declare a variable to store a reference to the instance of the ob-ject `Calculator`.

The next version of the class `CalculatorEngine` will declare a one-argument constructor. It'll take an argument of type `Calculator`.

JVM executes the constructor of the `CalculatorEngine` instance during instantiation of this class in memory. The `Calculator` object instantiates `CalculatorEngine` and passes to the engine's constructor a reference to itself:

```
CalculatorEngine calcEngine = new CalculatorEngine (this);
```

This reference contains the location of the calculator's instance in memory. The engine's constructor can store the value from the variable `this` in its own variable, say `parent`, and eventually use it in the method `actionPerformed()` to access the calculator's display field.

Attention, Bad Practice!

The variable parent in the following listing serves as a bridge from the object `CalculatorEngine` to `Calculator`. And the easy way to access `Calculator`'s `displayField` from `CalculatorEngine` is this:

```
parent.displayField.getText();
...
parent.displayField.setText(dispFieldText + clickedButtonLabel);
```

These two lines were taken from the code sample in **Listing 9-5**. This code works, but it violates one of the principles of object-oriented programming — encapsulation. The problem is that code from `CalculatorEngine` has direct knowledge of the internals of another object — `Calculator`. The engine "knows" that there is a field in `Calculator` called `displayField`, and it gets and sets its value directly.



Listing 9-5: Bad execution of object communication

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JButton;
```

```
public class CalculatorEngine implements ActionListener {

 Calculator parent; // a reference to the Calculator

 // Constructor stores the reference to the
 // Calculator window in  the member variable parent
 CalculatorEngine(Calculator parent){
   this.parent = parent;
 }

 public void actionPerformed(ActionEvent e){
   // Get the source of this action
   JButton clickedButton =  (JButton) e.getSource();

   // Get the existing text from the Calculator's
   // display field. Reaching inside another object is bad.
   String dispFieldText = parent.displayField.getText();

   // Get the button's label
   String clickedButtonLabel = clickedButton.getText();

   parent.displayField.setText(dispFieldText +
                               clickedButtonLabel);
 }
}
```

Imagine that for whatever reason you decide to use in `Calculator` something other than the `JTextField` widget to display the results of calculations. That other widget may not even have such APIs as `setText()` and `getText()`. Now you'll need to modify not only the `Calculator` class but also the code of the `CalculatorEngine` to replace the part that displays or reads the `displayField`. This is not the right way to design interactions between objects.

A Better Solution with a Public API

If `Calculator` needs to communicate with other objects, it should expose a public API to get or set data, but hide details about its internals. The class `Calculator` from Listing 8-4 declares widgets without using any access level qualifiers, so default package access level is applied. Hide these UI components as shown below by using the keyword `private`:

```
private JPanel windowContent;
private JTextField displayField;
private JButton button0;
private JButton button1;
...
```

Now `CalculatorEngine` won't be able to access `displayField` directly as it did in **Listing 9-5**. Defining public **getter** and **setter** methods in `Calculator` will al-

low outsiders to access `displayField` without knowing it exists. **Listing 9-6** demonstrates how a small change can protect data and enforce encapsulation.



Listing 9-6: Adding public API to Calculator

```
public class Calculator{
    private JTextField displayField;

    public void setDisplayValue(String val){
        displayField.setText(val);
    }

    public String getDisplayValue() {
        return displayField.getText();
    }

    // The rest of the code goes here
}
```

Now if you decide to replace the `JTextField` widget with another one, only the methods `setDisplayValue()` and `getDisplayValue()` will need a change; the `CalculatorEngine` won't need to be touched. **Listing 9-7** shows the proper way to access `Calculator` from the `CalculatorEngine`.



Listing 9-7: Using public API of Calculator

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JButton;

public class CalculatorEngine implements ActionListener {

 Calculator parent; // a reference to the Calculator

 // Constructor stores the reference to the
 // Calculator window in  the member variable parent
 CalculatorEngine(Calculator parent){
   this.parent = parent;
 }

 public void actionPerformed(ActionEvent e){
   // Get the source of this action
   JButton clickedButton =  (JButton) e.getSource();
```

```
    // Get the existing text from the Calculator's
    // display field. Reaching inside another object is bad.
    String dispFieldText = parent.getDisplayValue();

    // Get the button's label
    String clickedButtonLabel = clickedButton.getText();

    parent.setDisplayValue(dispFieldText +
                                    clickedButtonLabel);
  }
 }
```

More Swing Listeners

JDK comes with a number of event listeners located in the package `java.awt`. Here are some of them:

- Focus listener is notified when a widget gains or loses focus (for example, we say that a text field has focus if it has a blinking cursor).
- Item listener reacts to the selection of items from a list or a drop-down box.
- Key listener responds to user keypresses.
- Mouse listener responds to mouse clicks or the cursor's hovering over a widget.
- Mouse movement listener tells you if the mouse is being moved or dragged. To **drag** means to move the mouse while holding its left button down.
- Window listener gives you a chance to catch the moments when the user opens, closes, minimizes, or activates the window.

Table 9-1 shows the names of selected listener interfaces and the methods declared in these interfaces. For example, `FocusListener` declares two methods: `focusGained()` and `focusLost()`. This means that even if your class is interested only in knowing when a particular field gains focus, you also must include the empty method `focusLost()`. Java provides special adapter classes for each listener to spare you from having to manually code empty methods enforced by listener interfaces.

For more details on the listeners supported by Swing, refer to Oracle's Java tutorial at **http://java.sun.com/docs/books/tutorial/uiswing/events/eventsandcomponents.html**.

Table 9-1: Selected Swing Listeners

Interface	Methods to Implement
`FocusListener`	`focusGained(FocusEvent)` `focusLost(FocusEvent)`
`ItemListener`	`itemStateChanged(ItemEvent)`
`KeyListener`	`keyPressed(KeyEvent)` `keyReleased(KeyE` `keyTyped(KeyEvent)`
`MouseListener`	
`MouseMotionListener`	
`WindowListener`	

How to Use Adapters

Swing adapters are classes with implemented empty functions required by listener interfaces. Let's say you need to save some data on the disk when the user closes the window. According to **Table 9-1**, a class that implements the `WindowListener` interface has to include seven methods. This means that you'll have to write the code that saves the data in the method `windowClosing()` and also include six empty methods.

The package `java.awt` includes a number of adapter classes that implement corresponding listener interfaces, such as `KeyAdapter` and `WindowAdapter`. Instead of implementing `WindowListener` in a class that handles the window's events, just extend a class from `WindowAdapter` and override only the methods you are interested in, for example the method `windowClosing()`:

```
class MyWindowEventProcessor extends java.awt.WindowsAdapter {

  public void windowClosing(WindowEvent e) {
    // your code that saves the data goes here.
  }
}
```

The rest is easy: Register this class as an event listener in your GUI class (e.g. `Calculator`), as shown in **Listing 9-8**. Java offers adapters only for those listeners that declare more than one method.

You can register multiple listeners by using adapters in `Calculator`. For example, to allow the user to enter a number by pressing numeric keys, create a class based on `KeyAdapter`, instantiate it, and register it with `Calculator` too.



Listing 9-8: Registering adapter-based listener

```
MyWindowEventProcessor myWindowListener =  new MyWindowEventProcessor();
addWindowListener(myWindowListener);
```

You can achieve the same result using so-called anonymous inner classes, as explained in the next section.

Inner Classes

A class defined inside another is called an **inner** class. **Listing 9-9** shows an example of the class `TaxOptimizer` declared inside the class `Tax`. The class `TaxOptimizer` is a member inner class and has access to all variables of the class `Tax`. Placing one class inside another is just a way of saying that the classes belong together. After compilation, the class `Tax` file will produce two output files: `Tax.class` and `Tax$TaxOptimizer.class`.



Listing 9-9: Tax class including an inner TaxOptimizer class

```
class Tax{
    double grossIncome;
    int dependents;

  double calcStateTax(){

        TaxOptimizer tOpt = new  TaxOptimizer();
        return tOpt.optimize(grossIncome, dependents);
  }

      TaxOptimizer getTaxOptimizer(){
```

```
            return new TaxOptimizer();
        }

        class TaxOptimizer{

         int taxCode;

          void setTaxCode(int tCode){
                taxCode=tCode;
         }

         int optimize(double grossIncome, int dep){
            // Some optimization code goes here
              return 0;
         }
   }
}
```

An inner class defined as `static` can access only `static` variables of the outer class. The inner class can even be defined inside a method of an outer class. In this case this local inner class is available only when the outer method is called, and it can access only `static` variables of the top-level class.

The method `getTaxOptimizer()` in **Listing 9-9** returns an instance of the inner class if external classes need it. For example, if the class `TestTax` needs to access the method `setTaxCode()` from the inner class, it could do so as follows:

```
Tax t = new Tax(2, "NY", 50000);
Tax.TaxOptimizer tOptimizer = t.getTaxOptimizer();
tOptimizer.setTaxCode(12345);
```

Here's another syntax producing the same result:

```
Tax t = new Tax(2, "NY", 50000);
Tax.TaxOptimizer tOptimizer = t.new TaxOptimizer();
tOptimizer.setTaxCode(12345);
```

Anonymous Inner Classes

If an inner class does not have a name, it's called **anonymous**. The use of anonymous inner classes is pretty easy to understand in examples of implementing Java adapters. You've learned by now that using adapters is a

three-step process: Extend the adapter class, instantiate it, and register it as an event listener (see **Listing 9-8**). With anonymous inner classes you can perform all three steps in one shot, as in **Listing 9-10**.



Listing 9-10: Using anonymous class as adapter

```
this.addWindowListener(new WindowAdapter() {
           public void windowClosing(WindowEvent e) {
                  System.exit(0);
            }
         }
  );
```

Imagine that this code is placed in the `Calculator` class. The method `addWindowListener()` requires a subclass of `WindowAdapter` and the section in bold in **Listing 9-10** demonstrates the syntax of declaring an anonymous class that extends `WindowAdapter` and overrides the method `windowClosing()`.

The `new` operator instantiates the adapter, and because this is done inside the parentheses of the method `addWindowListener()` the newly created object is used as its argument. This adapter's implementation doesn't have a name, and we don't need to know its name in this context, do we? The instance of the adapter was created and registered as an event listener, and this is all that matters.

Closures

I've included the section on inner classes in the lesson about handling UI events just because it was easy to explain their practical use here, but inner classes were not created just for the handling of UIs. In some programming languages (such as JavaScript and ActionScript) there's a construct called **closure** — you can instantiate a method from a class, and the instantiation is done in such a way that the method can "remember" the data values from the class variables at the moment of instantiation.

At the time of this writing Java 7 has not been released yet, but there's a pretty good chance that closure will be one of the new features. Closures have simpler syntax than inner classes, and they're a long-anticipated addition to the language.

Try It

The goal of this lesson is to complete the code of the calculator. It has to look as in **Figure 9-2** and implement the functionality of all the buttons.

Figure 9-2



Lesson Requirements

For this lesson you should have Java installed.



You can download the code and resources for this Try It from the book's web page at www.wrox.com. You can find them in the Lesson9 folder in the download.

Step-by-Step

1. Create a new Eclipse project called Lesson9 and copy the `Calculator` from Lesson 8 into it (see Listing 8-4).

2. Create all missing UI components — use **Figure 9-2** as a prototype.

3. Create the event listener `CalculatorEngine` — all event processing and calculations should be performed there.

4. From `Calculator`, pass to the `CalculatorEngine` engine a reference to itself.

5. Register with `CalculatorEngine` all GUI components that can generate events.

6. Implement the code for the following scenario:

a. The user enters all the digits of the first number.

b. If the user hits one of the action buttons (+, -, /, or *), this indicates that the first number has been entered. Store this number and selected action in class variables (declare them first) and erase the number from the display text field. You'll need to convert the `String` value to `double` with the help of class `Double`.

c. The user enters the second number and hits the button =.

d. Convert the `String` value from the text field into a numeric type `double` so it is able to store numbers with a decimal point. Perform the selected action using this value and the number stored in the numeric variable from Step b.

e. Display the result in the display field and store this value in the variable that was used in Step b, for future calculations.

f. Run the calculator. If it works, give it to your friends.
To get the sample database files you can download Lesson 9 from the book's website at **www.wrox.com**.



Please select Lesson 9 on the DVD with the print book, or watch online at **www.wrox.com/go/fainjava** to view the video that accompanies this lesson.

Chapter 10

Introduction to Java Applets

Java was born in 1995, shortly after the Internet became available to the public. The Netscape web browser enabled regular people to go online and download content from remote servers. This content was mainly some formatted text mixed with static images. But the Java programming language provided a way to incorporate downloadable applications into the web pages!

A programmer needed to write a Java application called an **applet** in a special way using GUI classes from the `java.awt` package (Swing was not invented yet). Applets were deployed on the web servers along with the HTML files that included references to the applets. Web browsers understand HTML, the markup that consists of special **tags** placed in text files so web browsers can properly format and lay out web pages. HTML has a special tag, `<applet>`, that tells the browser where to find and how to size and align a Java applet.

You can download a Java applet to your computer from the Internet as part of a web page, and web browsers use the JVM plug-in to run applets. The version of JVM included with a web browser is very important, because if you're using, say, Java Swing classes, but the user's web browser has a really old JVM, the browser won't even know how to run the code that your applet uses.

In this lesson you'll learn the basics of applet creation and the applet's life cycle. In Lesson 11 you'll build a tic-tac-toe game and will deploy it

on the web server so other people can enjoy it.

An Unofficial History of Java Applets

Fifteen years ago Netscape's market share was over 90 percent, but the situation changed when Microsoft introduced Internet Explorer. Back in 1998 there was an infamous lawsuit between Microsoft and Sun Microsystems — the former started quietly introducing its own class library to Java, breaking the write-once-run-anywhere principle of Sun. Sun won that lawsuit. But as the saying goes, it won the battle but lost the war.

Microsoft stopped including the current version of JVM with Internet Explorer. I'm not blaming them for this, but it hurt applets' popularity — the end user couldn't just open a web page to see an applet that required, say, JVM version 1.3. The user would need to first download the proper version of JVM, and then only the applets that were written for JVM 1.3 would work. The process of downloading the right JVM plug-in was a multi-step process.

Sun Microsystems showed a lack of competence on the client side, specifically in the area of integration of JVM into web browsers, and Adobe Flash Player, which is also a virtual machine) dominates in this area now.

High penetration of the required run-time environment and the ability to upgrade it easily are crucial for any web-based technology. Adobe Flash Player shines in this area. Flash Player is a virtual machine with a small memory footprint (1.5 Mb), and its installation takes less than 20 seconds after one button click. For years the JVM plug-in was about 16 Mb, and installation was complicated for a non-programmer — it's a lot smaller now.

This situation changed after the release of Java 6 update 10, which includes the so-called next-generation Java plug-in. Now applets don't

run in the JVM packages with the web browser, but in a separate JVM launched by the Java plug-in. The applet still appears in the web browser's window, but now it doesn't depend on the willingness of the browser vendor to include the latest plug-in. You can read more about this next-generation Java plug-in at **https://jdk6.dev.java.net/plugin2**.

To control the settings of this plug-in in Microsoft Windows, use the Java Control Panel — select Start⇒⇒ Control Panel⇒⇒ Java. Under the Advanced tab find the item Java Plug-in where Enable the Next-generation Java Plug-in should be selected by default.

If you run Intel-based MAC OS X 10.6 or above, find out what Java plug-ins are installed by typing `about:plugins` into the address bar of your browser. To use the next-generation plug-in on Apple computers follow the instructions at **http://blogs.sun.com/thejavatutorials/entry/enabling_the_next_generation_java**.

The other major change introduced in the next-generation Java plug-in is the ability to launch Java applets directly from JNLP (Java Network Launch Protocol) files, which in the previous releases were used only in Java Web Start technology, which allowed local deployment of the applications over the network. As of Java 10.6.10 you can use the JNLP meta-descriptors to launch applets too. JNLP support is described in detail at **www.oracle.com/technetwork/java/javase/index-142562.html**.

What's the next step in applets' evolution? You'll be creating a UI for web-based applications with a relatively new language called JavaFX, which will be demonstrated in Lesson 37.

Restrictions of Java Applets

People browse the Internet without knowing if web pages contain Java applets or not, but they want to be sure that the files located in their computers will not be harmed by some bad guys who have added a

nasty applet to a page. **Unsigned applets** are the ones that haven't gone through a special **signing process** to obtain a security certificate indicating that they come from a trusted source. Unsigned (aka **untrusted**) Java applets operate inside a so-called **security sandbox** and have the following main restrictions:

- They can't access local files on your disk.
- They can connect only to the computer they were downloaded from.
- They can't start any other programs located in your computer.

Applets can be signed by VeriSign or Thawte and operate outside the security sandbox and be unrestricted in their ability to access the user's computer if the user accepts the applet's security. A signed applet will run outside the security sandbox. If the user refuses to accept the certificate, the applet will run within the security sandbox as unsigned.

To run an applet you'll need a Java class written in a special way, an HTML text file that contains the tag `<applet>` pointing to this class, and a web browser that supports Java. You can also test applets in Eclipse or using a special program called appletviewer. But before learning how to create applets, spend 15 minutes to get familiar with some HTML tags.

Learning HTML on the Run

Imagine that you've already written and compiled the game applet class called `TicTacToe`. It has to be deployed on the web servers along with the HTML wrapper — the file that includes the proper references to `TicTacToe.class`. HTML stands for Hypertext Markup Language. It defines a set of tags placed in angle brackets that can be inserted in a text file with any plain text editor, such as Notepad, TextEdit, or the editor that comes with Eclipse (select File ⇒ New ⇒ File). In this section I'll show you a bare minimum of HTML tags to get you going with applets,

but you can find a complete HTML tutorial at www.w3schools.com/html/html_primary.asp.

Let's start with creating the text file called `TicTacToe.html`. HTML files have names that end with `.html` or `.htm`. Inside they usually have two sections: header and body. HTML tags are placed inside angle brackets and most of them have the matching closing tags that start with a forward slash, for example `<head>` and `</head>`. The content of the very basic HTML file `TicTacToe.html` is shown in **Listing 10-1**.

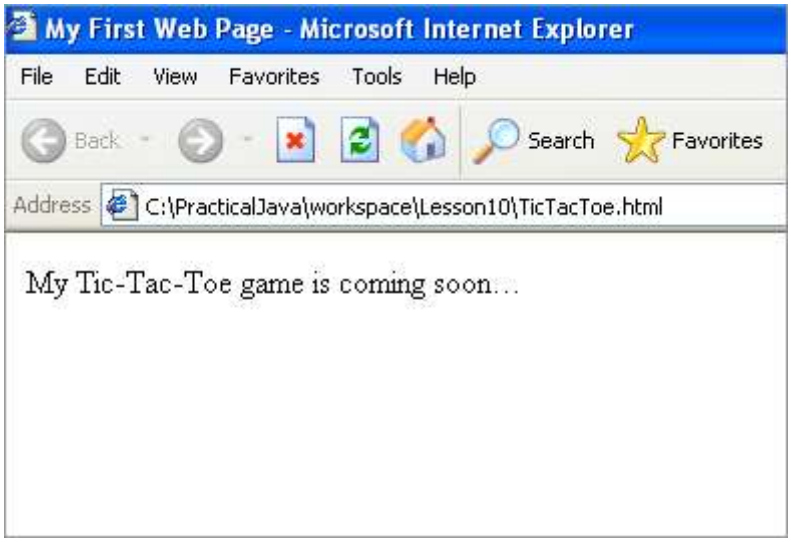


Listing 10-1: The basic HTML file

```
<html>
  <head>
   <title>My First Web Page</title>
  </head>
  <body>
     My Tic-Tac-Toe game is coming soon...
  </body>
</html>
```

You can place tags either in the same line, as we did with the tags `<title>` and `</title>`, or in separate lines. Open this file in your web browser using File ⇒ Open. As shown in **Figure 10-1**, the title bar of the window will read "My First Web Page," and inside the page you'll see the words "My Tic-Tac-Toe game is coming soon..."

Figure 10-1



Earlier versions of the HTML standard included the tag `<applet>` to incorporate Java applets into an HTML page. But as of HTML 4.01 this tag has been deprecated and you should use the tag `<object>` for Internet Explorer or `<embed>` for other web browsers. But Oracle, the company behind Java, recommends using `<applet>` for web pages accessed through the Internet and `<object>` or `<embed>` for intranet-based websites. The main reason is that web browser vendors are infamous for being inconsistent in implementing HTML standards, and it's safer to use the tried-and-true `<applet>` tag to give a predictable look and feel to the web pages that host Java applets. But an intranet is a more controlled environment. You can read about embedding applets with all of these tags at http://download.oracle.com/javase/6/docs/technotes/guides/plugin/developer_guide/using_tags.html, but in this tutorial I'll stick to the `<applet>` tag.

The deployment options for applets have been extended as of the release of Java 6 update 10. To learn all the options for applet deployment in depth, refer to the Oracle document Java Rich Internet Applications Deployment Advice, available at http://download.oracle.com/javase/6/docs/technotes/guides/jweb/deployment_advice.html.

Let's add the `<applet>` tag to the file `TicTacToe.html`, as shown in **Listing 10-2**. With some HTML tags you can specify additional attributes. The following are the main attributes of the `<applet>` tag:

- `code`: Contains the name of the applet Java class.
- `width`: Contains the width in pixels of the rectangular area on the screen that will be used by the applet.
- `height`: Contains the height of the area to be used by the applet.
- `codebase`: Specifies the root directory of your applet class relative to the location of the HTML file. For example, if the applet class is located in the sub-directory `bin`, specify `"./bin"` in the `codebase` attribute to avoid getting `FileNotFoundException`.
- `archive`: Use this attribute to specify the location of the `jar` file if your multi-class applet has been deployed as one archive file, which can be created with the help of the `jar` program located in the `bin` directory of your Java install.

The `<applet>` tag from **Listing 10-2** means that a compiled version of the Java class `TicTacToe` is located in the same directory as `TicTacToe.html`, and it'll occupy a screen area of 300 pixels by 250 pixels. Note that if an HTML tag doesn't have nested tags, placing a forward slash in front of the closing angle bracket is a replacement for a closing `</applet>` tag.

But in general, the `<applet>` tag may have nested elements. In particular, if you need to pass parameters to an applet you can do it via the nested element `<param>`.

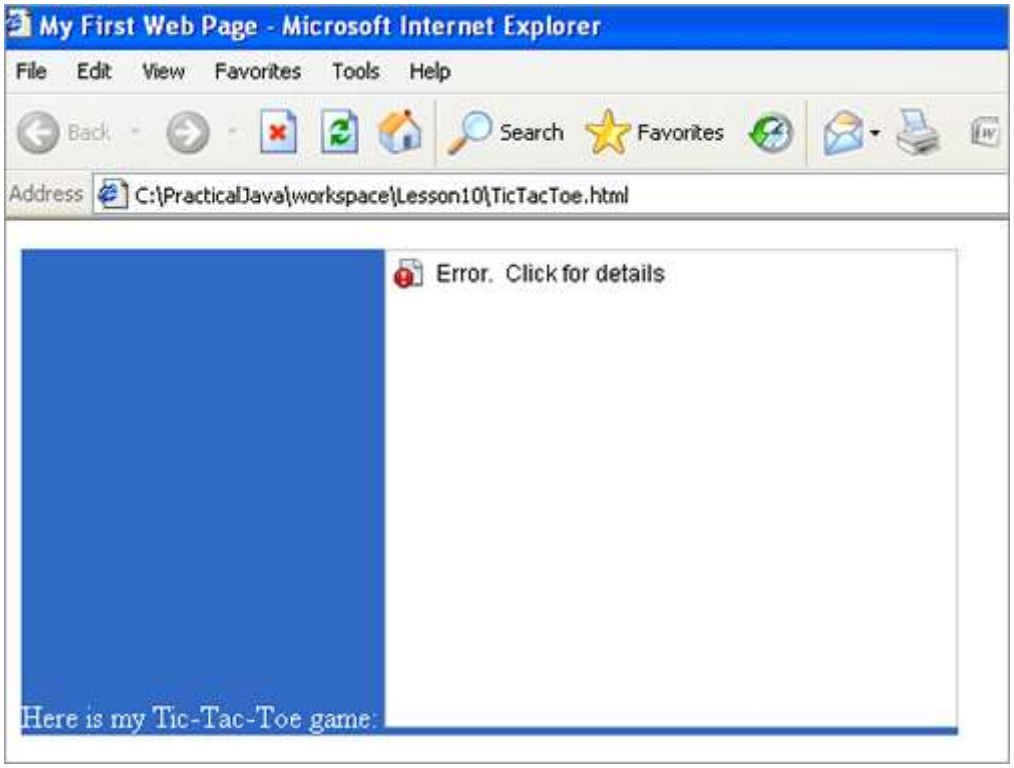


Listing 10-2: The basic HTML file

```
<html>
<body>
  Here is my Tic-Tac-Toe game:
  <applet code="TicTacToe.class" width=300 height=250 />
</body>
</html>
```

Now the screen looks different — instead of seeing the tic-tac-toe game you'll get an error message, as in **Figure 10-2**. No wonder: The web browser could not find `TicTacToe.class`, so it just shows a rectangle of 300 pixels by 250 pixels. You'll fix this error by creating the `TicTacToe` class a little later in this lesson.

Figure 10-2



Writing Applets Using Swing

Until the Swing library came into the picture, you could create Java applets only using UI components from the `java.awt` package. Even after Swing components were introduced, using them was somewhat problematic in applets downloadable over the Internet — what if the applet had to be run in an older browser that supported only Java 1.1 and didn't know about Swing classes? But modern applets use the new plug-in and don't depend on what JDK came with the web browser. In this book I'll be writing my applets with Java Swing.

Swing applets have to be inherited from the class `javax.swing.JApplet`. For example

```
class TicTacToe extends javax.swing.JApplet {
}
```

Unlike Java applications, applets do not need a `main()` method because the web browser will download and run them as soon as it sees the `<applet>` tag in the web page. The Java plug-in will also notify the applet when some important life-cycle events happen — for example, when the applet is starting, repainting, and so on.

If your applet needs to react to these events, you should program one of the **callback methods**: `init()`, `start()`, `paint()`, `stop()`, and `destroy()`. The Java plug-in will call these methods in the following cases:

- `init()` is called when the applet is loaded by the Java plug-in. It's called only once, playing a role similar to that of constructors in regular Java classes.
- `start()` is called right after `init()`. It is also called each time a user returns to the web page with the applet after visiting another page.
- `paint()` is called when the applet's window needs to be displayed or refreshed after some activity on the screen, for example, when the applet is overlapping with some other window and the browser needs to repaint it. This method will get the Graphics object as an argument.
- `stop()` is called when the user leaves the web page containing the applet.
- `destroy()` is called when the browser destroys the applet. Write code in this method only if the applet uses some other resources — for example, if it holds a connection to the remote computer it was downloaded from.

Even though you do not have to program all these methods, each applet must have at least `init()` or `paint()`. For example, `init()` is the right place to get some external parameters that may be passed to the applet. Say that the HTML file has an `<applet>` tag that looks like this:

```
<applet>
      <param>name="backgroundColor" value="0xc0c0c0"</param>
</applet>
```

To get the value of the parameter `backgroundColor` just write one line in the `init()` method:

```
String backgroundColor = getParameter("backgroundColor");
```

Listing 10-3 shows the code of the applet that displays the words "Hello World." This applet has only one method, `paint()`, that receives an instance of the `Graphics` object from the Java plug-in. The class `java.awt.Graphics` has several methods for painting. Our applet uses the method `drawString()` to draw the text "Hello World."



Listing 10-3: Hello World applet

```
import javax.swing.JApplet;
import java.awt.Graphics;

public class HelloWorld extends JApplet {
```

```
        public void paint(Graphics g){
             g.drawString("Hello World", 50, 100);
        }
  }
```

Create this class in Eclipse. Then run it as a Java Applet (Eclipse will use appletviewer to display it). The window appears as shown in **Figure 10-3**.

Figure 10-3



To test this applet in a web browser create the file `Hello.html` as you created the file in **Listing 10-2**. Start your web browser and open the file `Hello.html` using the browser's File and Open menus. You'll see the same output as in **Figure 10-3**.

Do you think that after this simple example you are ready to write a game program? You bet! You'll do it in Lesson 11. For a more detailed tutorial on Java applets, please visit the following web page: **http://download.oracle.com/javase/tutorial/deployment/applet/index.html**.

Try It

Turn the calculator from Lessons 8 and 9 into an applet.

Lesson Requirements

You should have Java installed.



You can download the code and resources for this Try It from the book's web page at www.wrox.com. You can find them in the Lesson10 folder in the download.

Step-by-Step

1. Create a new Eclipse project called Lesson10.

2. Take the `Calculator` class that you created in the Try It section of Lesson 8 and copy it into the project Lesson10. Give it an ancestor, `JApplet`.

3. Get rid of the `main()` method. Now the `Calculator` class will be instantiated for you by the Java plug-in inside the web browser.

4. Rename the `Calculator`'s constructor method into the applet's `init()` method.

5. Do not control the size of the calculator by calling `pack()`. Now the size of your `Calculator` is defined by the width and height attributes of the tag `<applet>`.

6. Do not instantiate `JFrame`.

7. Add the code to perform calculations that you completed at the end of Lesson 9.

8. Don't try to control the life cycle of the calculator by processing events of the `WindowListener`, as was explained in Lesson 9. Now your window lives inside the web browser.

9. Create the file `Calculator.html` and open it in the web browser. You should see your calculator.



Please select Lesson 10 on the DVD with the print book, or watch online at **www.wrox.com/go/fainjava** to view the video that accompanies this lesson.

Chapter 11

Developing a Tic-Tac-Toe Applet

Now that you're familiar with the basics of Java applets, it'll be nice to write a program a bit more useful than Hello World. In this lesson you'll program the game of tic-tac-toe as a Java Swing applet.

Every game implements some algorithm — a set of rules or a strategy that has to be applied depending on the player's actions. The algorithms for the same game can be simple or very complicated. When you hear that the world chess champion plays against a computer, he actually plays against a program. Teams of experts are trying to invent sophisticated algorithms to beat him. Let's come up with the algorithm for tic-tac-toe.

The Strategy

In the unlikely event that you aren't familiar with tic-tac-toe, or if it has a different name in your part of the world, read the following Wikipedia article: http://en.wikipedia.org/wiki/Tic-tac-toe. Our version of the popular game will implement the following strategy:

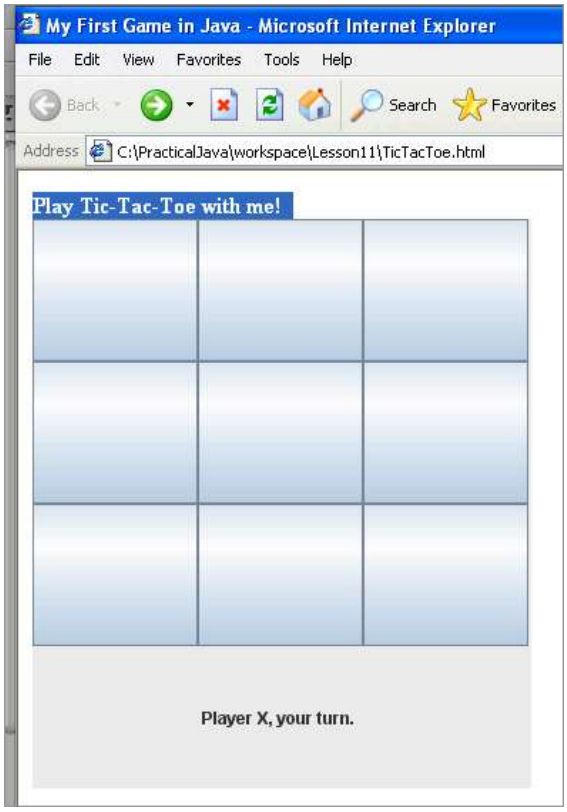
- The game will be played on a two-dimensional board of three cells by three cells.
- Two players can play this game. One will play with the symbol X, and the other will use O.
- The winner must have a full row, column, or diagonal of his or her designated symbol.
- After each move, the program has to check if there is a winner.
- The game should end if there are no empty squares left.
- The player has to press the New Game button to play again.
- When the program makes a decision about where to put the next letter O, it has to try to find a row, column, or diagonal that already contains two Os, and add the third accordingly.
- If there is no such row, column, or diagonal, the program has to try to find two Xs in such a configuration and place an O in the free cell to block the player's winning move.
- If no winning or blocking move was found, the program has to try to occupy the central square, or pick the next empty square at random.

Coding Tic-Tac-Toe

Let's implement the strategy for tic-tac-toe as an applet. First, program the UI portion, and then add event processing. I decided to use `JPanel`, contain-

ing nine `JButton`s, to render the three-by-three grid. Review the code in **Listing 11-1**. Note that the panel uses `GridLayout`, but with an extra row (four by three). The last row will display the player's identifier: "Player X" or "Player 0." **Figure 11-1** shows how it may look.

Figure 11-1



By setting the button label to X or O, on the click event, it's easy to keep track of each player's move. At the bottom of the window you'll see the name of the player who is to make the next move (see the method `setPlayerName()` in **Listing 11-1**). The method `reset()` will clear all the squares after the winner has been announced.

Available for download on Wrox.com

Listing 11-1: Coding UI for tic-tac-toe

```
package com.practicaljava.lesson11;

import java.awt.BorderLayout;
import java.awt.Component;
import java.awt.Font;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.GridLayout;

import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JLabel;
```

```
import javax.swing.JOptionPane;
import javax.swing.SwingConstants;

public class TicTacToeApplet1 extends JApplet{

    private static String PLAYERX = "Player X";
    private static String PLAYERO = "Player O";

    private String playerName = PLAYERX;

    private JButton button1;
    private JButton button2;
    private JButton button3;
    private JButton button4;
    private JButton button5;
    private JButton button6;
    private JButton button7;
    private JButton button8;
    private JButton button9;
    private JLabel playerNumber;
    private java.awt.Panel buttonsPanel;

    public void init(){

        initComponents();
    }

    private void initComponents(){

        buttonsPanel = new java.awt.Panel();
        button1 = new JButton();
        button2 = new JButton();
        button3 = new JButton();
        button4 = new JButton();
        button5 = new JButton();
        button6 = new JButton();
        button7 = new JButton();
        button8 = new JButton();
        button9 = new JButton();
        playerNumber = new JLabel(playerName, SwingConstants.CENTER);

       Font buttonFont = new Font("Times New Roman", Font.PLAIN, 60);

        button1.setFont(buttonFont);
        button2.setFont(buttonFont);
        button3.setFont(buttonFont);
        button4.setFont(buttonFont);
        button5.setFont(buttonFont);
        button6.setFont(buttonFont);
        button7.setFont(buttonFont);
        button8.setFont(buttonFont);
        button9.setFont(buttonFont);

        buttonsPanel.setLayout(new GridLayout(4, 3));

        buttonsPanel.add(button1);
        buttonsPanel.add(button2);
        buttonsPanel.add(button3);
        buttonsPanel.add(button4);
        buttonsPanel.add(button5);
        buttonsPanel.add(button6);
        buttonsPanel.add(button7);
        buttonsPanel.add(button8);
        buttonsPanel.add(button9);

        buttonsPanel.add(new Component(){});
        setPlayeName(PLAYERX);
        buttonsPanel.add(playerNumber);
```

```
        add(buttonsPanel);
    }

    private void setPlayerName(String playerName){

        this.playerName = playerName;
        playerNumber.setText(playerName +", your turn.");
    }

    public void reset(){
        button1.setText("");
        button2.setText("");
        button3.setText("");
        button4.setText("");
        button5.setText("");
        button6.setText("");
        button7.setText("");
        button8.setText("");
        button9.setText("");
        setPlayerName(PLAYERX);
    }
}
```

As you learned in Lesson 10, the callback method `init()` will be automatically called by the Java plug-in when the applet is downloaded by the web browser. While in development mode, you can also test your applets right in Eclipse with appletviewer. Note that the code in **Listing 11-1** does not set the size of the game board — that has to be done externally, either in an HTML file, as in **Listing 11-2**, or in Eclipse in the Run Configuration window. The width of my game board is set to 350 pixels and the height is 400.



Listing 11-2: HTML wrapper for the tic-tac-toe applet

```
<HTML>
  <HEAD>
   <Title>My First Game in Java</Title>
  </HEAD>

  <BODY>

     <STRONG>Play Tic-Tac-Toe with me! </STRONG>
     <BR>
     <APPLET code="com.practicaljava.lesson11.TicTacToeApplet"
             codebase="./bin" width="350" height="400" />

  </BODY>

</HTML>
```

Swing and GroupLayout

The code in **Listing 11-1** was written manually. But most likely, if you're developing a complex GUI, you'll use some Swing GUI builder such as Matisse that will allow you to design windows in what-you-see-is-what-you-get mode by dragging and dropping visual components onto the design area. If you look at the auto-generated code you may find `GroupLayout`, created by the GUI builder (but you can program this layout manually, too).

`GroupLayout` groups components inside a container. Imagine a shipping or billing form that contains several text labels, one under another, each having an input field next to it. Each label contains a different number of characters, which means that you'll need to align the input fields so the form looks pretty. If you can think of this UI as of two vertical groups (labels belong to one and input fields to another), you can simply set the gap between the left and right groups to 20 pixels or so.

Similarly, you can distinguish some horizontal groups in your UI. If you want to learn more about `GroupLayout`, follow this link: **http://download-llnw.oracle.com/javase/6/docs/api/javax/swing/GroupLayout.html**.

After the UI part is ready, let's take care of the processing logic. Our applet will be processing mouse clicks to set the label of the selected button to X or O and checking for the winner. That's why you'll need to perform these steps:

1. Add `implements MouseListener` to the class `TicTacToeApplet`.

2. Add `MouseListener` to each of the buttons so they properly report which player (X or O) clicked them, for example

```
button1.addMouseListener(this);
```

3. Implement methods required by the `MouseListener` interface — only the `mouseClicked()` callback will have code in this applet.

4. Call the method `checkForWinner()` on each click to determine whether there are three identical symbols in a row, column, or diagonal, as shown in **Listing 11-3**.



Listing 11-3: Processing button click events

```
public void mouseClicked(MouseEvent e) {

  JButton currentButton = (JButton)e.getComponent();
  if (currentButton.getText() == ""){
         if (playerName == PLAYERX) {
                currentButton.setText("X");
```

```
                setPlayerName(PLAYERO);

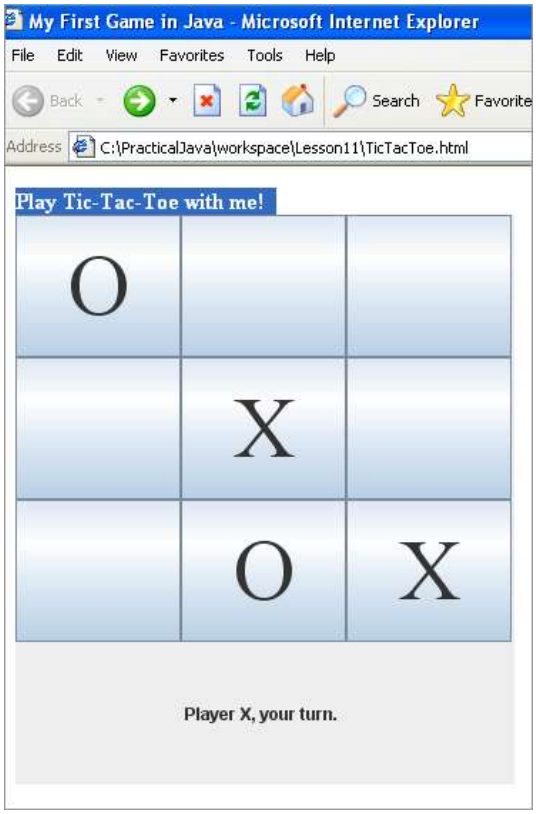
            } else if (playerName == PLAYERO){

                currentButton.setText("O");
                setPlayerName(PLAYERX);
            }
    }

          checkForWinner();
}
```

When the user clicks a button, the method `mouseClicked()` checks the name of the current player and displays X or O accordingly by calling `currentButton.SetText()`. **Figure 11-2** depicts how this looks in the web browser.

Figure 11-2



The method `checkForWinner()` will call another private method, `findThreeInARow()`. If three identical symbols are found diagonally or in any vertical or horizontal line, the method `checkForWinner()` will congratulate the winner and call `reset()` so the new game can be started. The methods `checkForWinner()` and `findThreeInARow()` are shown in Listings **11-4** and **11-5** respectively.

Available for download on Wrox.com

Listing 11-4: The method checkForWinner()

```
private void checkForWinner(){

  String [] str = {"OK"};

  if(findThreeInARow()){

   String winnerName=(playerName == PLAYERX)?PLAYERO:PLAYERX;
   JOptionPane.showOptionDialog(this, winnerName.concat(
             " won!!! Congratulations!!!"), "Congratulations!",
JOptionPane.YES_OPTION, JOptionPane.PLAIN_MESSAGE, null, str, "OK");

   reset();
  }
}
```



Listing 11-5: The method findThreeInARow()

```
private boolean findThreeInARow(){

 if ((button1.getText() == button2.getText() && button2.getText() ==
   button3.getText() && button1.getText() != "") ||
 (button4.getText() == button5.getText() && button5.getText() ==
   button6.getText() && button4.getText() != "") ||
 (button7.getText() == button8.getText() && button8.getText() ==
  button9.getText() && button7.getText() != "") ||
 (button1.getText() == button4.getText() && button4.getText() ==
   button7.getText() && button1.getText() != "") ||
 (button2.getText() == button5.getText() && button5.getText() ==
   button8.getText() && button2.getText() != "") ||
 (button3.getText() == button6.getText() && button6.getText() ==
  button9.getText() && button3.getText() != "") ||
 (button1.getText() == button5.getText() && button5.getText() ==
   button9.getText() && button1.getText() != "") ||
 (button3.getText() == button5.getText() && button5.getText() ==
   button7.getText() && button3.getText() != "")
)
   //There is a winner!
  return true;

else
  //No three-in-a-row was found
  return false;
}
```

The complete code for this version of tic-tac-toe is included on the website and is given in **Listing 11-6** for your reference.



Listing 11-6: Complete code for the tic-tac-toe game

```
package com.practicaljava.lesson11;
import java.awt.BorderLayout;
import java.awt.Component;
import java.awt.Font;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JOptionPane;
import javax.swing.SwingConstants;
public class TicTacToeApplet extends JApplet                                    implements MouseListener {

    private static String PLAYERX = "Player X";
    private static String PLAYERO = "Player O";
    private String playerName = PLAYERX;
    private JButton button1;
    private JButton button2;
    private JButton button3;
    private JButton button4;
    private JButton button5;
    private JButton button6;
    private JButton button7;
    private JButton button8;
    private JButton button9;
    private JLabel playerNumber;
    private Panel buttonsPanel;

    public void init(){
        initComponents();
    }
    private void initComponents(){
        buttonsPanel = new Panel();
        button1 = new JButton();
        button2 = new JButton();
        button3 = new JButton();
        button4 = new JButton();
        button5 = new JButton();
        button6 = new JButton();
        button7 = new JButton();
        button8 = new JButton();
        button9 = new JButton();

        playerNumber = new JLabel(playerName,                                SwingConstants.CENTER);
        button1.addMouseListener(this);
        button2.addMouseListener(this);
        button3.addMouseListener(this);
        button4.addMouseListener(this);
        button5.addMouseListener(this);
        button6.addMouseListener(this);
        button7.addMouseListener(this);
        button8.addMouseListener(this);
        button9.addMouseListener(this);

        Font buttonFont = new Font("Times New Roman",                                    Font.PLAIN, 60);
        button1.setFont(buttonFont);
        button2.setFont(buttonFont);
        button3.setFont(buttonFont);
        button4.setFont(buttonFont);
        button5.setFont(buttonFont);
        button6.setFont(buttonFont);
        button7.setFont(buttonFont);
        button8.setFont(buttonFont);
        button9.setFont(buttonFont);

        buttonsPanel.setLayout(new java.awt.GridLayout(4, 3));
        buttonsPanel.add(button1);
        buttonsPanel.add(button2);
```

```
        buttonsPanel.add(button3);
        buttonsPanel.add(button4);
        buttonsPanel.add(button5);
        buttonsPanel.add(button6);
        buttonsPanel.add(button7);
        buttonsPanel.add(button8);
        buttonsPanel.add(button9);
        buttonsPanel.add(new Component(){});

        setPlayerName(PLAYERX);
        buttonsPanel.add(playerNumber);
        add(buttonsPanel);
    }

    private void setPlayerName(String playerName){
        this.playerName = playerName;
        playerNumber.setText(playerName  + ", your turn. ");
    }

    private void reset(){
        button1.setText("");
        button2.setText("");
        button3.setText("");
        button4.setText("");
        button5.setText("");
        button6.setText("");
        button7.setText("");
        button8.setText("");
        button9.setText("");

        setPlayerName(PLAYERX);
    }

   private void checkForWinner(){
        String [] str = {"OK"};
       if(findThreeInARow()){
         String winnerName=         (playerName == PLAYERX)?PLAYERO:PLAYERX;

         JOptionPane.showOptionDialog(this,          winnerName.concat(" won!!! Congratulations!!!"),
         "Congratulations!", JOptionPane.YES_OPTION,          JOptionPane.PLAIN_MESSAGE, null, str, "OK");
            reset();
        }
}

public void mouseClicked(MouseEvent e) {
        JButton currentButton = (JButton)e.getComponent();
        if (currentButton.getText() == ""){
            if (playerName == PLAYERX) {
                currentButton.setText("X");
                setPlayerName(PLAYERO);
            } else if (playerName == PLAYERO){
                currentButton.setText("O");
                setPlayerName(PLAYERX);
            }
        }
        checkForWinner();
    }

      public void mousePressed(MouseEvent e) {}
      public void mouseReleased(MouseEvent e) {}
      public void mouseEntered(MouseEvent e) {}
      public void mouseExited(MouseEvent e) {}

      public boolean findThreeInARow(){
          if ((button1.getText() == button2.getText() && button2.getText() == button3.getText() && button1.getTe
                (button4.getText() == button5.getText() && button5.getText() == button6.getText() && button4.get
              (button7.getText() == button8.getText() && button8.getText() == button9.getText() && button7.getTe
             (button1.getText() == button4.getText() && button4.getText() == button7.getText() && button1.getText
             (button2.getText() == button5.getText() && button5.getText() == button8.getText() && button2.getText
```

```
            (button3.getText() == button6.getText() && button6.getText() == button9.getText() && button3.getText
            (button1.getText() == button5.getText() && button5.getText() == button9.getText() && button1.getText
            (button3.getText() == button5.getText() && button5.getText() == button7.getText() && button3.getText
        ){
             return true;
        }else {
              return false;
        }
      }

}
```

Congratulations! You've completed your first game in Java. This program uses a simple strategy because our goal is just to learn how to program Java applets. There is one more important class for working with Swing UI using the Swing Worker Thread, but it's a bit too early to explain it here because I haven't covered Java threads yet. You can read about the `SwingWorker` class in Lesson 20.

Try It

Today's homework is to modify the code from **Listing 11-6** to make it shorter and change the game — the current version of tic-tac-toe requires two human players, but I want you to replace one of them with the computer (which also has some brainpower). I also want you to highlight the winning combination rather than displaying a pop-up window with congratulations.

Lesson Requirements

You should have Java installed.



You can download the code and resources for this Try It from the book's web page at **www.wrox.com**. You can find them in the Lesson11 folder in the download.

Step-by-Step

1. Create a new Eclipse project called Lesson11, copy `TicTacToeApplet` from the DVD accompanying the book into this project, and run it to ensure that you can play this two-player game.

2. The code in **Listing 11-6** performs lots of repetitive actions on buttons, such as `setFont()`, `setText()`, and `addMouseListener()`. Create an array and store all instances of `JButton` there. Introduce loops to perform all repetitive actions on the buttons.

3. Get rid of the pop-up window congratulating the winner with `JOptionPane.showOptionDialog()`. Display this text instead at the bottom of the window (use the existing `JLabel playerNumber` for this).

4. Highlight the winning combination by giving the winning squares (Swing `JButton`s) a different color.

5. Add a new button, Play Again, at the bottom of the window for starting a new game.

6. Replace the moves of Player O with randomly generated moves made by the computer right after the Player X moves. Use the method `nextInt()` from the class `java.util.Random`. If the randomly generated square is occupied, call this method again.

7. Test this program and enjoy the game!

8. This step is optional. If you want to make the computer's moves smarter, learn **minimax strategy** (see http://en.wikipedia.org/wiki/Minimax) to enable the computer to select the best move.



Please select Lesson 11 on the DVD with the print book, or watch online at www.wrox.com/go/fainjava to view the video that accompanies this lesson.

Chapter 12

Developing a Ping-Pong Game

The tic-tac-toe game developed in Lesson 11 was a simple one. My goal was to show you the basic layout of UI components and event processing. Now you'll learn how to draw and move such objects as ovals, rectangles, and lines in a window. You'll also learn how to process keyboard events. To add a little fun to these subjects, you'll be learning all these things while creating a ping-pong game. This game will have two players: the user and the computer.

The Strategy

Let's come up with the rules for our version of the ping-pong game:

1. The game lasts until one of the players (the user or the computer) reaches a score of 21. The computer's racket is on the left.

2. The user's racket movements will be controlled by the mouse.

3. The game score will be displayed at the bottom of the window.

4. A new game starts when a player presses the N key on the keyboard; the Q key ends the game; S serves the ball.

5. Only the user can serve the ball; the computer can't.

6. For one of the players to win a point the ball must go beyond the opponent's racket's vertical line when the racket is not blocking the

ball.

7. When the computer bounces the ball, the ball can move only horizontally to the right.

8. If the ball contacts the user's racket in the upper half of the table, the ball should be moving up and left. If the ball was located in the bottom part of the table, it should move down and left.

It sounds like a very complicated task, but let's split it into a set of simple tasks and solve them one by one. The first version of the game will have only some of these rules implemented: It'll just paint the table, move the racket, and display the coordinates of the mouse pointer when you click the mouse button.

The Code

From the code organization perspective, you need to decide how many classes you need to write to separate UI and functionality. This game will consist of two classes and one interface:

- The class `PingPongGreenTable` will take care of the visual part. During the game it'll display the table, rackets, and ball. The final version of the table is shown in **Figure 12-1**. This book doesn't show colors, but the table surface is green, the rackets are blue and yellow, and the ball is red.

Figure 12-1



- The class `PingPongGameEngine` will be responsible for calculating the ball's and rackets' coordinates, starting and ending the game, and serving the ball. The engine will pass the current coordinates of the UI components to `PingPongGreenTable`, which will repaint itself accordingly.
- The interface `GameConstants` will contain declarations of all constants that the game needs, for example the width and height of the table, the starting positions of the rackets, and so on.

The User Interface

The first version of the game will do only three things:

- Display the green ping-pong table.
- Display the coordinates of the mouse pointer when you click the mouse.
- Move the user's racket up and down according to the mouse pointer's movements.

Listing 12-1 shows the first version of the UI. The class `PingPongGreenTable1` is a subclass of the Swing `JPanel`.



Listing 12-1: Coding UI of the ping-pong game (version 1)

```
public class PingPongGreenTable1 extends JPanel implements GameConstants1{

    JLabel label;
    public Point point = new Point(0,0);

    private int ComputerRacketX =15;
    private int kidRacketY =KID_RACKET_Y_START;

    Dimension preferredSize = new Dimension(TABLE_WIDTH,TABLE_HEIGHT);

    // This method sets the size of the frame. It's called by JVM
    public Dimension getPreferredSize() {
          return preferredSize;
    }

    // Constructor instantiates the engine
    PingPongGreenTable1(){

      PingPongGameEngine1 gameEngine = new PingPongGameEngine1(this);

       // Listen to mouse clicks to show its coordinates
         addMouseListener(gameEngine);

       // Listen to mouse movements to move the rackets
         addMouseMotionListener(gameEngine);
     }

     // Add a panel with a JLabel to the frame
     void addPaneltoFrame(Container container) {

        container.setLayout(new BoxLayout(container, BoxLayout.Y_AXIS));
        container.add(this);

         label = new JLabel("Click to see coordinates");
         container.add(label);
      }

      // Repaint the window. This method is called by the JVM
      // when it needs to refresh the screen or when a
      // method repaint() is called from PingPointGameEngine1

       public void paintComponent(Graphics g) {

         super.paintComponent(g);
         g.setColor(Color.GREEN);

         // paint the table green
```

```
            g.fillRect(0,0,TABLE_WIDTH,TABLE_HEIGHT);

            // paint the right racket
            g.setColor(Color.yellow);
            g.fillRect(KID_RACKET_X_START,kidRacketY,5,30);

           // paint the left racket
            g.setColor(Color.blue);
            g.fillRect(ComputerRacketX,100,5,30);

            //paint the ball
            g.setColor(Color.red);
            g.fillOval(25,110,10,10);

            // Draw the white lines on the table
            g.setColor(Color.white);
            g.drawRect(10,10,300,200);
             g.drawLine(160,10,160,210);

            // Display the dot as a small 2x2 pixels rectangle
           if (point != null) {
               label.setText("Coordinates (x,y): " +
                             point.x + ", " + point.y);
        g.fillRect(point.x, point.y, 2, 2);
    }
        }

        // Set the current position of the kid's racket
        public void setKidRacketY(int xCoordinate){

         this.kidRacketY = xCoordinate;
       }

       // Return the current position of the kid's racket
        public int getKidRacketY(int xCoordinate){
          return kidRacketY;
       }

         public static void main(String[] args) {

            JFrame f = new JFrame("Ping Pong Green Table");

            // Ensure that the window can be closed
           // by pressing a little cross in the corner
            f.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);

            PingPongGreenTable1 table = new PingPongGreenTable1();

            table.addPaneltoFrame(f.getContentPane());
```

```
        // Set the size and make the frame visible
        f.pack();
        f.setVisible(true);
    }
}
```

At any given time the game needs to know the exact coordinates of the mouse pointer. The constructor of `PingPongGreenTable1` will create an instance of the event listener class `PingPongGameEngine`, which will perform some actions when the user clicks the mouse button or just moves the mouse.

The method `addPaneltoFrame()` creates a `JLabel` component that will display the coordinates of the mouse.

This class is not an applet, which is why instead of the method `paint()` it uses the method `paintComponent()`. This method is called by JVM when it needs to refresh the window, or when our program calls the method `repaint()`. In other words, the method `repaint()` internally calls `paintComponent()` and provides our class with the object `Graphics` so it can paint on the window. Always call this method after recalculating the coordinates of the rackets or the ball to display them in the proper position.

To paint a racket, set the color first, and then fill a thin vertical rectangle with this paint using the method `fillRect()`. This method needs to know the x- and y-coordinates of the top left corner of the rectangle and its width and height in pixels.

The ball is painted with the method `fillOval()`, which needs to know the coordinates of the center of the oval, its height and its width. When the height and width of the oval are the same, it is a circle.

The x-coordinate in a window increases from left to right, and the y-coordinate increases from top to bottom (see **Figure 12-2**). For exam-

ple, the width of the rectangle shown in **Figure 12-1** is 100 pixels, and the height is 70.

Figure 12-2



The x- and y-coordinates of the corners of this rectangle are shown in parentheses.

Another interesting method is `getPreferredSize()`. The program creates an instance of the Swing class `Dimension` to set the size of the ping-pong table. The JVM needs to know the dimensions of the window, which is why it calls the method `getPreferredSize()` of the `PingPongGreenTable1` object. This method returns to the JVM an object, `Dimension`, that is created in the code according to the size of our table.

The first version of the UI is shown in **Figure 12-3**. You can move the mouse, and the correct racket moves along. When you click the mouse anywhere on the table, the current coordinates are displayed at the bottom.

Figure 12-3



The Constants

Both table and engine classes use some constant values that do not change. For example, the class `PingPongGreenTable1` uses the width and height of the table, and `PingPongGameEngine1` needs to know the ball movement increments — the smaller the increment, the smoother the movement. It's convenient to keep all the constants (`final` variables) in an interface. In our game the name of the interface is `GameConstants1`. If a class needs these values, there are two ways of making them available in the game classes.

The first one is just adding the clause `implements GameConstants1` to the class declaration and using any of the final variables from this interface as if they had been declared in the class itself. This method of adding constants has a code readability drawback — it won't be easily understood where these final variables were declared. Both the table and engine classes (see **Listing 12-3**) implement the `GameConstants1` interface shown in **Listing 12-2**.

The second (and better) way of using these constants is to import the interface in the classes that need them, and use the fully qualified names of the constants, e.g. `GameConstants1.TABLE_WIDTH`. Changing the code of **Listing 12-3** to apply this technique is one of the assignments in the Try It section.

If you decide to change the size of the table, ball, or rackets you'll need to do it in only one place — the `GameConstants1` interface.



Listing 12-2: The game's constants (version 1)

```
public interface GameConstants1 {

 int TABLE_WIDTH =  320;
 int TABLE_HEIGHT = 220;
 int KID_RACKET_Y_START = 100;
 int KID_RACKET_X_START = 300;
 int TABLE_TOP = 12;
 int TABLE_BOTTOM = 180;
 int RACKET_INCREMENT = 4;
}
```

A program cannot change the values of these variables, because they were declared as `final`. But if, for example, you decide to increase the size of the table, you'll need to change the values of `TABLE_WIDTH` and `TABLE_HEIGHT` and then recompile the `GameConstants1` interface.

The Engine

The decision maker in this game is the class `PingPongGameEngine1`, which implements two mouse-related interfaces. The first version of the engine is shown in **Listing 12-3**.



Listing 12-3: The game's engine (version 1)

```
public class PingPongGameEngine1  implements
       MouseListener, MouseMotionListener, GameConstants1{

 PingPongGreenTable1 table;
 public int kidRacketY = KID_RACKET_Y_START;

 // Constructor. Stores a reference to the UI object
 public PingPongGameEngine1(PingPongGreenTable1 greenTable){
   table = greenTable;
 }

// Methods required by the MouseListener interface
 public void mouseClicked(MouseEvent e) {

   // Get X and Y coordinates of the mouse pointer
   // and set it to the "white point" on the table
   // Bad practice! Fix it in the Try It section
       table.point.x = e.getX();
table.point.y = e.getY();

  //The method repaint internally calls the table's
  // method paintComponent() that refreshes the window
  table.repaint();
 }

 public void mouseReleased(MouseEvent e) {};
 public void mouseEntered(MouseEvent e) {};
 public void mouseExited(MouseEvent e) {};
 public void mousePressed(MouseEvent e) {};

 // Methods required by the MouseMotionListener interface
 public void mouseDragged(MouseEvent e) {}

 public void mouseMoved(MouseEvent e) {
  int mouseY = e.getY();

  // If a mouse is above the kid's racket
  // and the racket did not go over the table top
  // move it up, otherwise move it down

  if (mouseY < kidRacketY && kidRacketY > TABLE_TOP){
kidRacketY -= RACKET_INCREMENT;
  }else if (kidRacketY < TABLE_BOTTOM) {
       kidRacketY += RACKET_INCREMENT;
  }

  // Set the new position of the racket table class
```

```
            table.setKidRacketY(kidRacketY);
            table.repaint();
        }
    }
```

The `MouseListener` will have code only in the method `mouseClicked()`. On every mouse click this method will draw a small white dot on the table and display its coordinates. This code is not very useful for our game, but I wanted to show you a simple way to get the coordinates of the mouse from the `MouseEvent` object that is given to the program by JVM.

The method `mouseClicked()` sets the coordinates of the variable `point` depending on where the mouse pointer was when the player pressed the mouse button. After coordinates are set, this method asks the JVM to repaint the table.

`MouseMotionListener` reacts to movements of the mouse over the table, and we'll use its method `mouseMoved()` to move the user's racket up or down.

The method `mouseMoved()` calculates the next position of the user's racket. If the mouse pointer is above the racket (the t-coordinate of the mouse is less than the y-coordinate of the racket) this method ensures that the racket will not go over the top of the table.

When the constructor of the table class creates the engine object, it passes to the engine a reference to the table's instance (the keyword `this` represents a reference to the memory location of the object `PingPongGreenTable1`). Now the engine can "talk" to the table, for example by setting the new coordinates of the ball or repainting the table if needed.

I've included an example of a bad programming practice in **Listing 12-3**. The engine class is directly accessing the variable point declared in another class, `PingPongGreenTable1`. It would be better to make the variable `point` private and to provide a public method in the class `PingPongGreenTable1`

to set the coordinates of the point. This way if you decide to rename the variable `point` you won't need to change the engine's class; the public interface will remain the same. You'll need to fix this bad practice in the "Try It" section.

In our game rackets move vertically from one position to another using the four-pixel increment defined in the interface `GameConstants1` (the engine class implements this interface). For example, the next line subtracts four from the value of the variable `kidRacketY`:

```
kidRacketY -= RACKET_INCREMENT;
```

Suppose the y-coordinate of the racket is 100: after this line it becomes 96, which means that the racket has to be moved up. You can get the same result using the following syntax:

```
kidRacketY = kidRacketY - RACKET_INCREMENT;
```

Finishing the Ping-Pong Game

Well, to finish the game you need to learn about multi-threading, which will be explained in Lesson 20. If you feel a craving to finish the game right now, read the first half of Lesson 20 and then examine the code that comes on the accompanying DVD and read the following explanations. I won't include the full listing of the completed game here — just some highlights to help you in reading the code located on the disk. The class names in the completed version of the code are the same as the ones used in this lesson, but they do not end with the digit 1.

Let's start with the class `PingPongGreenTable`. You don't need to display a white dot when the user clicks the mouse; this was just an exercise to show you how to get the coordinates of the mouse pointer. That's why I'll remove the declaration of the variable `point` and the lines that paint the white dot from the method `paintComponent()`. Also, the constructor does not need to add the `MouseListener` anymore; it was there only to display the point's coordinates.

On the other hand, `PingPongGreenTable` should react to some of the keyboard buttons (N for new game, S for serving the ball, and Q for quitting the game). Adding `KeyListener` to the engine will take care of the communication with the keyboard. The declaration of the engine will look like this:

```
public class PingPongGameEngine implements Runnable,
       MouseMotionListener, KeyListener, GameConstants{...}
```

Accordingly, you'll need to add to the constructor of the class `PingPongGreenTable` the following line:

```
addKeyListener(gameEngine);
```

To make the code a little more encapsulated, I've also moved the `repaint()` calls from the engine class to `PingPongGreenTable`. Now this class will be responsible for repainting itself when needed.

I've also added methods to change the positions of the ball and the computer's racket, and to display messages. The final `GameConstants` interface has more variables than the one shown in **Listing 12-2**. You

should be able to guess what they are for just by looking at their names.

Following is a summary of the changes I've made in the class `PingPongGameEngine` — review the code from the DVD and identify these changes:

- I have removed the interface `MouseListener` and all its methods, because we're not processing mouse clicks anymore. `MouseMotionListener` will take care of all mouse movements.
- This class now implements the `Runnable` interface, and you can find the decision-making code in the method `run()`. Look at the constructor: I create and start a new thread there. The method `run()` implements the game strategy rules in several steps, and all these steps are programmed inside the `if` statement `if(ballServed)`, which is a short version of `if(ballServed==true)`.
- Note the use of the conditional `if` statement that assigns a value to the variable `canBounce` in the code marked in the comments as Step 1. The variable will get a value of either `true` or `false` depending on the result of the following expression:

```
canBounce = (ballY >= computerRacketY &&
                    ballY < (computerRacket_Y + RACKET_LENGTH)? true: false);
```

- The class `PingPongGameEngine` implements the `KeyListener` interface, and the method `keyPressed()` checks what letter was keyed in — N to start, Q to quit, or S to serve. The code of this method enables the user to type either capital or lowercase letters, for example N or n.
- I've added several private methods like `displayScore()`, `kidServe()`, and `isBallOnTheTable()`. These methods are declared `private` because they are used within this class only, and other classes do not even have to know about them.
- Today's computers are too fast, and this makes the ball movements difficult to control. That's why I've slowed the game down by call-

ing a method called `Thread.sleep()`. A static method, `sleep()`, will pause this particular thread for a number of milliseconds given as an argument of this method (see Lesson 20 for details).

- To make the ball movement a little more difficult to predict, when the user's racket hits the ball it moves diagonally. That's why code changes not only the x-coordinate of the ball, but the y-coordinate as well.

That's all there is to it. Run the program and enjoy the game!

Try It

Your homework is to re-factor the code shown in this lesson. First split the UI, the constants, and the engine into separate Java packages. Review the code and add the proper access modifiers: Make the variables and methods that don't have to be exposed to other classes `private`. The code in **Listing 12-3** shows an example of a bad programming practice: The engine class directly modifies the internal variable `point` of the UI class. Fix this.

Lesson Requirements

You should have Java installed.



You can download the code and resources for this Try It from the book's web page at **www.wrox.com**. You can find them in the Lesson12 folder in the download.

Hints

After learning how to work with files in Lesson 16, consider adding a new feature to the ping-pong game — that of saving the score in a file and providing a menu option to show the past scores.

Step-by-Step

1. Create a new Eclipse project called Lesson12, copy the two classes whose names end with 1 from the DVD accompanying the book into this project, and run it to ensure that the ping-pong table is shown and the correct racket is moving up and down according to the movements of the mouse.

2. Create three packages named ui, engine, and common.

3. Move the class `PingPongGreenTable1` to ui, PingPongGameEngine1 to engine, and GameConstants1 to common.

4. Include the import statements where required.

5. Compile and run the program.

6. In the class `PingPongGreenTable1` introduce the public method `setPointCoordinates()`, which will set the x- and y-coordinates of the variable point (make this method private).

7. Browse the source code of all these classes and add the proper data access qualifiers to all variables and methods.

8. Refactor the code of `PingPongGameEngine1` to import the interface `GameConstants1` rather than implementing it.



Please select Lesson 12 on the DVD with the print book, or watch online at **www.wrox.com/go/fainjava** to view the video that accompanies this lesson.

Chapter 13

Error Handling

Fixing the compiler's errors becomes trivial as you become more comfortable with the Java syntax. But you also should ensure that your programs process run-time errors that may happen regardless of your proficiency with the language itself.

Let's say a Java program that reads customers' data is deployed in production. What's going to happen if this file gets corrupted? Will the program crash with a scary geeky error message, or will it stay alive, displaying a user-friendly message such as, "There seems to be a problem with the file customers. Please make sure that the file is not corrupted"? Error processing in the Java world is called **exception handling**, which is the subject of this lesson.

Stack Trace

When a Java application is running, the JVM performs a number of internal and application-specific method calls. If a run-time error occurs that's not handled by the program, the program prints a **stack trace**, which reflects in the call stack the sequence of unfortunate events that caused this error. A stack trace helps software developers follow the workflow of the program that led to the error.

To illustrate what a stack trace may look like, consider the program shown in **Listing 13-1**, which deliberately divides by zero.



Listing 13-1: Generating stack trace by dividing by zero

```
1 class TestStackTrace{
2    TestStackTrace()
3    {
4        divideByZero();
5    }
6
7    int divideByZero()
8    {
9        return 25/0;
10    }
11
12    static void main(String[]args)
13    {
14           new TestStackTrace();
15
16    }
17 }
```

Listing 13-2 depicts the output of this program, which has traced what happened in the program stack before the error occurred. Read the output from the last line upward. It shows that the program was executing the methods `main()`, `init()` for the constructor, and `divideByZero()`. The line numbers 14, 4, and 9, respectively, indicate where in the program these methods were called. After that the `ArithmeticException` was thrown — the code in line 9 tried to divide by zero. Turning the line numbers on in the Eclipse IDE helps you locate problematic code.



Listing 13-2: Sample stack trace

```
c:\temp>java TestStackTrace

     Exception in thread "main"
     java.lang.ArithmeticException: / by zero
        at TestStackTrace.divideByZero(TestStackTrace.java:9)
        at TestStackTrace.<init>(TestStackTrace.java:4)
        at TestStackTrace.main(TestStackTrace.java:14)
```

Executing any Java program means running multiple threads, as explained in Lesson 20, and the stack trace output reflects what was happening in the main thread of our simple `TestStackTrace` program.

Java Exceptions

In many programming languages, error processing depends on the programmer's goodwill and experience. Java forces a programmer to include the error-handling code for the majority of errors; otherwise the programs won't even compile.

Say you need to write a piece of code that reads a file containing data about customers. It's easy to foresee that unless the code includes error handling there is a chance that one day you'll see a stack trace instead of a customer list.

The creators of Java didn't want to allow this code to fail just because some programmers are too lazy to include error-handling code. Java forces you to place such code inside a `try/catch` block, as in **Listing 13-3**. Whenever you read or write files you have to process input/output (I/O) errors.



Listing 13-3: Catching I/O errors

```
try{
    fileCustomer.read();
}
catch (IOException e){
    System.out.println("There seems to be a problem with the file customers. ");
}
```

Read the code from **Listing 13-3** as follows: "Try to execute `fileCustomer.read()`, and if an error occurs, jump into the `catch` section and execute the code from there." `IOException` is a class that contains information about input/output errors.

In the case of an I/O error, the method `read()` **throws an exception** (for more details on reading files refer to Lesson 16). The `catch` block will catch this error and process it. The program won't terminate and this exception is considered handled. If the method `read()` finishes successfully, the code in the **section** `catch` won't be executed.

Exception Hierarchy

Errors in Java are represented as classes that can be divided into two major types: those that were caused by bad programming and those that were thrown because of some other external condition. For example, if a program declares a variable of type `Tax`, but this object was never instantiated, any attempts to call the non-static method `calcTax()` will result in `NullPointerException`:

```
Tax t;
t.calcTax();
```

This situation could have been predicted and properly handled by the programmer.

If a run-time error can be fixed, the exception is called **checked**. The method `reads()` from **Listing 13-3** will throw an exception and the JVM will try to find the code that handles this error. Such an exception can be anticipated and recovered from without the need to change the code. While the program remains operational, the user can find the missing file containing the list of customers and try again to populate the GUI with this list.

All exceptions are subclasses of `Throwable`, which has two immediate descendants: `Error` and `Exception`. Subclasses of `Exception` are called **checked** exceptions and have to be handled in your code. You can declare and throw your own application-specific exception, for example `BadShipmentAddressException`.

Subclasses of the class `Error` are fatal JVM errors and are called **unchecked**. You are not required to put them in `try/catch` blocks as there is not much you can do if, say the JVM runs out of memory and crashes.

How is a programmer supposed to know in advance if some Java method may throw a particular exception and that the `try/catch` block should therefore be used? No need to memorize anything. If a method throws an exception, the Java compiler will print an error message similar to this one:

```
"Tax.java":  unreported exception: java.io.IOException; must be caught or declared to be thrown at line 57
```

try/catch Blocks

There are five Java keywords that can be used for exception handling: `try`, `catch`, `finally`, `throw`, and `throws`. One `try` block can have multiple `catch` blocks, to provide handling for more than one type of error. For example, when a program tries to read a file, the file may not be there —

you must catch the `FileNotFoundException`. If the file is there, but the code tries to read past the end of file, the `catch` clause for `EOFException is necessary`. **Listing 13-4** illustrates a multi-`catch` block.



Listing 13-4: One try with multiple catch statements

```
public void getCustomers(){
   try{
         fileCustomers.read();
   }catch(FileNotFoundException e){
         System.out.println("Can not find file Customers");
   }catch(EOFException e1){
            System.out.println("Done with file read");
   }catch(IOException e2){
      System.out.println("Problem reading  file " +
                                          e2.getMessage());
   }
}
```

The order of the `catch` statements may be important if the exceptions being caught belong to the same inheritance branch. For example, the class `EOFException` is a subclass of the more generic `IOException`, and you have to put the `catch` block for the subclass first. If you place the `catch` block for `IOException` before the one for `EOFException`, the latter block will never be reached — the end-of-file errors will be intercepted by the `IOException catch` block. A lazy programmer would cut corners and handle the exceptions that may happen during the reading of the file in a coarse-grained way (the compiler won't complain):

```
public void getCustomers(){
    try{
         fileCustomers.read();
```

```
    }catch(Exception e){
         System.out.println("Problem reading  file " +
                                              e.getMessage());
    }
}
```

A `catch` block receives an instance of the `Exception` object that contains a short explanation of a problem, and the method `getMessage()` of the `Exception` object will return this info. If the description of an error returned by `getMessage()`is not clear enough, try the method `toString()` instead.

If you need more detailed information about the exception, use the method `printStackTrace()` on the received `Exception` object (see **Listing 13-6**). It will print all internal method calls that led to this exception, as described in the section “Stack Trace” earlier in this lesson.

The throws Clause

In some cases it makes more sense to handle an exception not in the method where it happened, but in the calling one. Let’s use the same example of code that reads a file. Because the method `read()` may throw an `IOException`, you should either handle it or declare that the calling method may throw it. The latter is done in **Listing 13-5**.



Listing 13-5: Using throw clause

```
class CustomerList{

    void getAllCustomers() throws IOException{

     // Some other code goes here
     // Don't use try/catch if you are not handling exceptions here
```

```
      file.read();
    }

  public static void main(String[] args){
      System.out.println("Customer List");

     // Some other code goes here

     try{
       // Since getAllCustomers()declared an exception,
       // either  handle  it over here, or re-throw it
       //(see the throw keyword explanation below)

         getAllCustomers();
     }
     catch(IOException e){
       System.out.println("Customer List is not available");
     }
  }
```

In **Listing 13-5** `IOException` has been propagated from the method `getAll-Customers()` to the `main()` method, and has been handled there.

The finally Clause

The code can exit the `try/catch` block in several ways:

- The code inside the `try` block successfully ends and the program continues.
- The code inside the `try` block runs into a `return` statement.
- The code inside the `try` block throws an exception and control goes to the `catch` block.

As you can see, in some cases only the code from the `try` block will work; in some cases part of the code from the `try` block and all the code in `catch` will be invoked. If there is a piece of code that must be executed regardless of the success or failure of the code in the `try` block, put it under the clause `finally`.



Listing 13-6: Using clause finally

```
try{
    file.read();
    //file.close();   don't close files inside try block
}
catch(Exception e){
    e.printStackTrace();
}
finally{

   file.close();

}
```

The code in **Listing 13-6** will definitely close the file regardless of the success of the `read` operation, because the `close()` function is called in the `finally` block. But if you place the `close()` function inside the `try` block without writing the `finally` portion, in case of error during read the program will jump from the failed `read()` into `catch`, skipping the `close` operation. The `finally` clause is typically used for the release of system resources.

If you are not planning to handle exceptions in the current method, they will be propagated to the calling one. In this case you can use the `finally` clause without the `catch` clause:

```
void myMethod () throws IOException{
     try{
         file.read();
      }
      finally{
         file.close();
```

```
      }
}
```

The throw Operator

If an exception has occurred in a method, you may want to catch the exception, do partial error processing such as error logging, or throw it to the calling method again for further processing or just to make the user aware of the problem. In some cases you may want to catch an exception and handle it but throw another exception (with modified error information) to the calling method.

The `throw` statement is used to throw Java exception objects. The object that a program throws must be `Throwable` (you can throw a ball, but you can't throw a grand piano). This means that you can throw only subclasses of the `Throwable` class and that all Java exceptions are its subclasses:

```
class CustomerList{

  void getAllCustomers() throws Exception{

    // some other code goes here

    try{
           file.read(); // this line may throw an exception
    } catch (IOException e) {

      // Log this error here, and rethrow another exception
       throw new  Exception ("Customer List is not available "+
                                          e.getMessage());
         }
   }

  public static void main(String[] args){
      System.out.println("Customer List");
        ...
     try{
```

```
        // Since the  getAllCustomers() declares an exception,
        // you have  to either  handle  or re-throw it
         getAllCustomers();
    }
    catch(Exception e){
          System.out.println(e.getMessage());
    }
 }
```

Creating Your Own Exceptions

Programmers can also create exceptions specific to their business applications. Such a class has to be a subclass of one of the classes from the hierarchy of `Throwable` objects.

Let's say you are in business selling bikes and need to validate a customer's order. Create a new class, `TooManyBikesException`, and throw it if someone tries to order more bikes than can fit into one container, as in the class `BikeOrder` shown in **Listing 13-7**.



Listing 13-7: Creating and throwing your own exceptions

```
class TooManyBikesException extends Exception{

   TooManyBikesException (String msgText){
        super(msgText);
   }
}

class BikeOrder{
   ...
  static  void validateOrder(String bikeModel,
             int quantity) throws TooManyBikesException{

// perform  some data validation, and if the entered
```

```
// the quantity or model is invalid, do  the following:

  throw new TooManyBikesException("Can not ship" +
         quantity+" bikes of the model " + bikeModel +);
 }
}

class OrderWindow extends JFrame{
     ...
 void actionPerformed(ActionEvent e){

   // the user clicked on the "Validate Order" button

   try{
      bikeOrder.validateOrder("Model-123", 50);

     // the next line will be skipped in case of  exception
     txtResult.setText("Order is valid");

   } catch(TooManyBikesException e){

      txtResult.setText(e.getMessage());

   }
 }
}
```

`TooManyBikesException` shown in **Listing 13-8** has a unique name and the text includes some information specific to the shipping business. But another way to provide application-specific information is to declare one or more additional variables in the custom exception. These variables carry multiple pieces of data that describe the erroneous situation and may help in fixing it.



Listing 13-8: Custom exception with extra property

```
class TooManyBikesException extends Exception{
```

```
    // Declare an application-specific property
    ShippingErrorInfo shippingErrorInfo;

    TooManyBikesException (String msgText,ShippingErrorInfo shippingErrorInfo){
         super(msgText);
         this.shippingErrorInfo = shippingErrorInfo;
    }
}
```

Listing 13-8 illustrates the code that adds an application-specific object, `ShippingErrorInfo`, to the custom `TooManyBikesException`. An application can prepare the object describing a shipping error and pass it as an argument to the constructor of the exception. The latter will store it in the class variable `shippingInfo`, and whatever method catches this exception can extract the `ShippingErrorInfo` object and act accordingly.

In distributed Java EE applications an exception can travel through several tiers (such as JMS, EJB, Servlet, Swing client), and not only does having a custom property inside the exception object ensure that the valuable information won't be lost, but each tier can add more specifics to this custom property, helping in tracing the error.

There is also a class called `RemoteException`, with a field called `detail`, that's used for reporting communication errors. You can extend this class to make remote exceptions more descriptive. This subject may be more appropriate for the lessons about the server-side technologies (Lesson 26 and above), but since this is the lesson dedicated to Exceptions, I mentioned it here.

Keep in mind that a Java program can communicate with non-Java systems, and that throwing `RemoteException` or its subclass is a proper way to handle cross-language errors. For example, this technique is used for communications based on Adobe's AMF protocol between server-side Java and a rich Internet client application written with Adobe Flex.

For more details on Java exceptions refer to Oracle's tutorial at http://java.sun.com/docs/books/tutorial/essential/exceptions.

Try It

Create a Swing application for placing bike orders. It has to have a drop-down list (`JComboBox`) containing several bike models, `JTextField` for entering quantity, and `JButton` for validating the order.

Make up several combinations of bike models and quantities that will throw an exception. Use the code snippets from **Listing 13-7** as examples. The validation should start when the user clicks the button to validate the order.

Lesson Requirements

You should have Java installed.



You can download the code and resources for this Try It from the book's web page at www.wrox.com. You can find them in the Lesson13 folder in the download.

Step-by-Step

1. Create a new Eclipse project, Lesson13.

2. Learn how to work with `JComboBox` at the following tutorial: http://java.sun.com/docs/books/tutorial/uiswing/components/combobox.html.

3. Process events and revalidate the order whenever the user selects a new bike model or changes the quantity of the order.

4. Throw and handle `TooManyBikesException` if the order can't be shipped.



Please select Lesson 13 on the DVD with the print book, or watch online at **www.wrox.com/go/fainjava** to view the video that accompanies this lesson.