The first element of an array in Java always has an index of `0`. While declaring an array you can place brackets either after the data type or after the variable name. Both of the following declarations are correct:

```
String friends[];
String[] friends;
```

You must know the size of the array before assigning values to its elements. If you want to be able to dynamically change the size of an array during the run time, consider other Java collection classes from the package `java.util`, such as `Vector` and `ArrayList`. Besides arrays Java has lots of collection classes that can store multiple related values, for example `HashMap`, `List`, and `LinkedList`. You'll have a chance to see their use in the code samples accompanying this book. **Listing 5-1** contains sample code that partially populates an array.




**Listing 5-1:** Populating a simple array

```
public class Girlfriends1 {

 public static void main(String[] args) {
       String []  friends = new String [20];  // Declare and instantiate array
       friends[0] = "Masha";                  //Initialize the first element
       friends[1] = "Matilda";                //Initialize the second element
       friends[2] = "Rosa";
       // ...
       friends[18] = "Hillary";
       friends[19] = "Natasha";

       System.out.println("The third girl's name is " + friends[2]);
       System.out.println("The twentieth girl's name is " + friends[19]);
```

```
 }
}
```

An array has a property length that “knows” the number of elements in the array. The next line shows how you can get this number:

```
int  totalElements = friends.length;
```

If you know all the values that will be stored in the array at the time of its declaration, you can declare and initialize an array at the same time. The following line declares, instantiates, and populates an array of four elements.

```
String []  friends = {"Masha", "Matilda", "Rosa", "Sharon"};
```

Our array `friends` is not as practical as a Rolodex, though — it does not store girls’ phone numbers. Luckily, Java supports multidimensional arrays. For example, to create a two-dimensional array (names and phone numbers), declare it with two sets of square brackets:

```
String  friends [][] = new String [20][2];
friends[0][0] = "Masha";
friends[0][1] = "732 111-2222";
friends[1][0] = "Matilda";
friends[1][1] = "718 111-2222";
...
```

```
friends[19][0] = "Sharon";
friends[19][1] = "212 111-2222"
```

### Loops

Loops are used to repeat the same action multiple times. When you know in advance how many times you want to repeat an action, use the `for` loop. Let's print the names from the one-dimensional array `friends`.

```
int  totalElements = friends.length;

for (int i=0; i < totalElements;i++){
   System.out.println("I love " + friends[i]);
}
```

The preceding code reads "Print the value of the element `i` from the array `friends` starting from `i=0` and incrementing `i` by one (`i++`) until `i` reaches the value equal to the value of `totalElements`." **Listing 5-2** adds a `for` loop to the program shown in **Listing 5-1**.




**Listing 5-2:** Looping through the array

```
public class Girlfriends2 {

       public static void main(String[] args) {
              String []  friends = new String [20];
              friends[0] = "Masha";
              friends[1] = "Matilda";
              friends[2] = "Rosa";
              friends[18] = "Hillary";
```

```
                friends[19] = "Natasha";

                int  totalElements = friends.length;
                int i;
                for (i=0; i<totalElements;i++){
                   System.out.println("I love " + friends[i]);
                }

        }
}
```

Since the `friends` array has been declared with a size of `20`, Java run time has allocated memory for 20 elements. But the code in **Listing 5-2** has populated only five of the 20 elements of the array, which explains why the output of this program looks as follows:

```
I love Masha
I love Matilda
I love Rosa
I love null
I love null
I love null
I love null
I love null
I love null
I love null
I love null
I love null
I love null
I love null
I love null
I love null
I love null
I love null
I love null
I love Hillary
I love Natasha
```

The keyword `null` represents an absence of any value in an object. Even though the size of this array is `20`, only five elements were initialized.

There's another syntax of the `for` loop, known as the **for-each loop**. You simply declare a variable of the same type as the objects stored in an array and specify the array to iterate. The following code snippet declares the variable `girl`, and the colon means "in." Read this loop's condition expression as "for each `element` in `friends`." This syntax allows you to not worry about checking the size of the array, and there is no need to increment any loop variable either. This is an elegant and short loop notation:

```
for (String girl: friends){
    System.out.println("I love  " + girl);
}
```

You can rewrite the program in **Listing 5-2** using the `while` loop, which is used when you do not know the exact size of the array, but just the condition of exit from the loop. Use the keyword `while`:

```
int  totalElements = friends.length;
int i=0;
while (i<totalElements){
   System.out.println("I love " + friends[i]);
       i++;    // the same as i=i+1;
}
```

Just think of a program that reads and processes the records from a database (see Lesson 22). When you write a Java program you don't know how many elements the database has, and even if you do know, this number can change in the future, so it's better to use loops with the exit condition than to use a hard-coded number of repetitions.

Use the keyword `break` to prematurely jump out of the loop on the line below the ending curly brace. For example, if you want to find the first `null` element in the `friends` array, write the following:

```
while (i<totalElements){
       if (friends[i]==null){
          System.out.println("The element " + (i+1) + " is null");
          break;
       }

      System.out.println("I love " + friends[i]);
      i++;
}  // closing curly brace for the loop
```

The `if` statement in the preceding code checks the value of each element of the array, and as soon as it finds `null`, the loop prints the message about it, stops the iteration process, and goes to the line below the closing curly brace of the loop, if any.

The keyword `continue` allows you to force the loop to jump up to its first line and retest the loop exit condition. The following code snippet will print only those values from the array that are not `null`.

```
while (i<totalElements){
    if (friends[i]==null){
       i++;
       continue;
    }

    System.out.println("I love " + friends[i]);
    i++;
                   }
```

```
    System.out.println("The iteration is over");
}
```

The preceding code uses an `if` statement you have not seen before, which allows you to check a certain condition and redirect the program to execute one or another portion of the code accordingly. In this case, if the loop runs into a `null` value (the double equals sign means "compare the values for equality"), it increments by one the value of the variable `i` and goes to the beginning of the `while` loop, skipping the rest of the code within the loop body. (Later in this lesson there's a section explaining the syntax of the various `if` statements in greater detail.) The complete code of the program, illustrating a `while` loop with a `continue` statement, is shown in **Listing 5-3**.




**Listing 5-3:** While loop with continue statement

```
public class WhileLoopDemo {

       public static void main(String[] args) {
              String []  friends = new String [20];
              friends[0] = "Masha";
              friends[1] = "Matilda";
              friends[2] = "Rosa";
              friends[18] = "Hillary";
              friends[19] = "Natasha";

              int  totalElements = friends.length;
              int i=0;

              while (i<totalElements){
                       if (friends[i]==null){
                                i++;
                                continue;
                        }

                        System.out.println("I love " + friends[i]);
```

```
                        i++;
                        }
                        System.out.println("The iteration is over");
        }
}
```

There is a rarely used `do-while` version of the `while` loop. It starts with the `do` keyword followed by the body of the loop in curly braces, and the `while` condition comes last. Such loop syntax guarantees that the code written in the body of the loop will be executed at least once, because the loop condition will be tested only after the first pass into the loop. In the following loop at the very minimum the statements about reading the array `friends` will be printed for each element of the array even if every one of them is `null`:

```
do {
   System.out.println("Reading the element" + i +" of array friends");
    if (friends[i]==null){
       i++;
       continue;
    }

    System.out.println("I love " + friends[i]);
    i++;
                    }
} while (i<totalElements);
```

The worst thing that can happen in any loop is a programmer's mistake in the loop exit condition that will always evaluate the loop condition as `true`. In programming this is known as **infinite loop**. To get a better feeling for what this term means, comment out the line that increments the value of the variable `i` inside the `if` statement, and your program will never end unless you forcefully stop it or your computer runs out of power. The reason is clear: If the program will enter the code block that just has the statement `continue`, the value of the variable

`i` will never increase and the loop execution condition `i<totalElements` will hold true forever.

### Debugging Java Programs

In programmer's lingo a **bug** is an error in a program that causes the program to work in an unexpected way. Don't confuse a bug with a syntax error in the code. The latter will be caught by the Java compiler before you even start the program, while bugs are your run-time enemies. To **debug** a program is to identify and fix the run-time errors in code.

The simplest way to debug a program is to print the value of the "suspicious variables" with `System.out.println()` or the like. You may think that a certain variable will get a particular value during the execution, but you might be wrong, and printing its value from the running program may reveal why your code produces unexpected results.

Java also comes with a logging API (application program interface) that allows you to **log** the run-time values in a file or other destination. Logging is out of the scope of this book, but you can find `Logger` and other supporting classes in the Java package `java.util.logging`.

You'll be using a debugger that comes with an IDE daily. Even though Lesson 2 was dedicated to Eclipse IDE, explaining debugging back then would have been a bit premature, as you didn't have much Java code to debug. Now you're ready to learn how the Eclipse debugger can help you.

Let's see how the `while` loop from **Listing 5-3** works by running the `WhileLoopDemo` program with the Eclipse debugger. You can do it either by right-clicking the name of the program and selecting the menu option Debug As Java Application or by clicking the little icon with an image of a bug in the Eclipse toolbar. But before doing this, set a breakpoint on the line of code where you want the program to pause execution

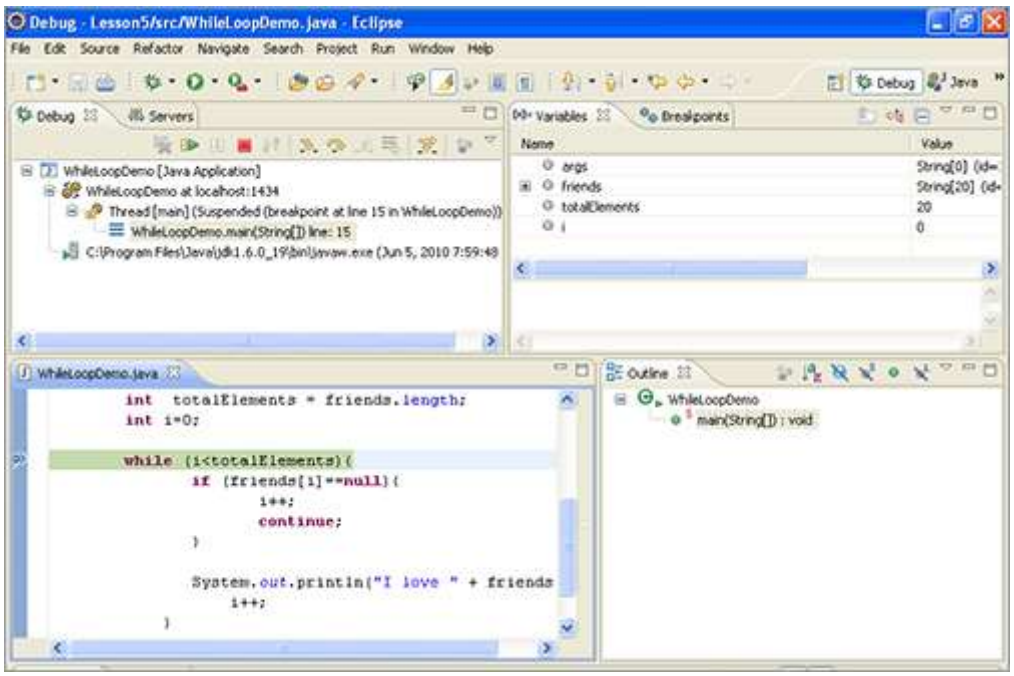
and allow you to start watching the program internals in the Eclipse Debugger's Perspective.

I'd like to pause this program right before it enters the `while` loop, so let's set the breakpoint on the following line:

```
while (i<totalElements){...}
```

Double-click the gray vertical bar located between the Package and Editor views — you should see an image of a little bullet there. This is a breakpoint, and if the program runs into the code with the set breakpoint, Eclipse IDE will switch to Debugger perspective and pause the program, highlighting the line that's about to execute. **Figure 5-1** shows how the Debugger perspective may look at this point.

**Figure 5-1**




Note the little arrow in the Editor view next to the bullet — it shows you the line of code that will be executed next. The Variables view shows you the values of the program variables at this moment. The

variable `i` is equal to `0`; the value of `totalElements` is `20`. If you click the little plus sign by the variable `friends`, it'll show the content of this array.

Figure 5-2

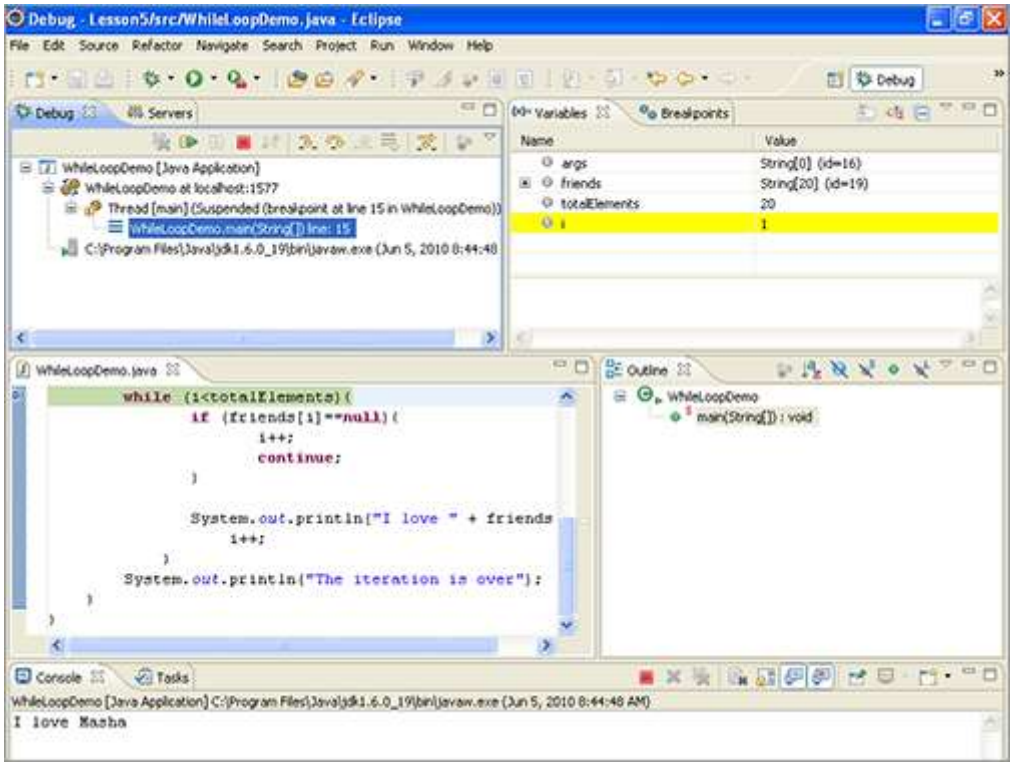



Now try to execute this program step-by-step, watching how the value of the variable `i` changes. You'll control program execution by clicking the buttons in the toolbar for the Debug view (see **Figure 5-2**).

The green play button means "Continue executing the program until it ends or hits another breakpoint." The red square button stops the debugging process. The first curvy yellow arrow (Step Into) is used if the code has been paused on a line that calls a method, and you'd like to debug the code of the method being called. The second curvy arrow (Step Over) allows you to execute the current line without stepping into any methods that may be called in this line of code.

There are other and less frequently used buttons on this toolbar, which you can study on your own, but let's enter the loop by clicking the Step Over button. Keep clicking this button and observe that the program doesn't enter the `if` statement, which is correct — the first element of the friends `array` is not `null`. Then `if` prints "I love Masha" in the console view, increments the value of the variable `i` to `1` (see the Variables view in **Figure 5-3**), and returns to the loop condition to check if the value of `i` is still less than the value of `totalElements`.

Figure 5-3




Keep clicking the Step Over button, and expand the variable `friends` in the Variables view to see its elements in order to better understand why the program skips or enters the `if` statement in the loop. Note the moment when the program exits the loop and prints the message "The iteration is over." After spending some time debugging the program you should appreciate the value of the Eclipse Debugger. In my opinion, the best way to learn a new programming language is to run someone else's code through a good debugger. The program may be bug-free, but this will help you to better understand its flow.

### More about if and switch Statements

If you need to change the execution of the program based on a certain condition you can use either `if` or `switch` statements, which were introduced in Lesson 3. In this section you'll see more flavors of the conditional statements.

### The Flavors of if Statements

In **Listing 5-3** you saw one version of the `if` statement:

```
if (friends[i]==null){
    i++;
    continue;
}
```

The curly braces must be used in the `if` statement to specify a block of code that has to be executed if the condition is true. Even though the curly braces are not required if there's only one statement in the code block, using them is considered a good practice — this makes the program more understandable for other people who may need to read your code (or for yourself six months from now).

```
if (friends[i]==null){
    System.out.println("I found an array element with null value");
}
// Some other code goes here
```

The code below the closing curly brace will always be executed, which may not be desirable. In this case use the `if-else` syntax:

```
if (friends[i]==null){
    System.out.println("I found an array element with null value");
} else{
    // Some other code goes here
}
```

In the preceding code snippet, the "some other code" will be executed only if the current element of the array is not `null` (or, as we put it in Java, `friends[i]!=null`). You can write an `if` statement with multiple `else`

clauses. For example, the following code will print only one love confession:

```
if (friends[i]==null){
    System.out.println("I found an array element with null value");
} else if (friends[i] == "Natasha"){
    System.out.println("I love my wife so much!");
} else if (friends[i] == "Masha"){
    System.out.println("I fell in love with Masha when I was in the 8th grade.");
} else{
    System.out.println("I used to love " + friends[i] + " at some point.");
}
```

Using conditional statements can be a lot more complex than comparing two values. You can build expressions using `boolean` AND, OR, and NOT operators. The AND operation is represented as `&&`, like this:

```
if (age<20 && gender=="male") {
      // do something only for males under 20 years old
}
```

For the OR condition use `||`, for example:

```
if (age < 30 || yearsInArmy > 0) {
      // do something with people yonger than 30 or those who served
      // in the army regardless of their age
}
```

For the NOT condition (aka negation) use the `!` sign:

```
boolean hasBachelorDegree;
// Read the person record and assign the value true or false
// to the variable hasBachelorDegree
if (!hasBachelorDegree) {
      // do something with people without bachelor degree
}
```

The negation symbol can be placed either in front of a `boolean` variable, as in the preceding code snippet, or in front of the expression that returns `boolean`. The next code example shows how to use negation Note that the comparison expression was placed inside the parentheses; otherwise the Java compiler would assume that you're trying to negate a `String` value called `friends[i]`, which is not allowed.

```
if (!(friends[i]=="Hillary")){
       System.out.println("Still looking for Hillary.");
}
```

Imagine a situation in which you need to assign a value to a variable, but the value depends on the result of a certain condition. To assign the variable in a verbose way you can use a regular `if` statement and test some condition, writing one assignment operator for a result of `true`, and another in the `else` section for a result of `false`. But if you use a special construct called a **conditional operator** (`?`), the same task can be done more simply.

The conditional operator consists of the following parts: an expression that returns `boolean`, a question mark, an expression that returns some value if the first expression evaluates to `true`, a colon, and an expression that returns a value to be used if the first expression returns `false`. It sounds complicated, but it's not. The following code sample shows

how to assign a $3,000 bonus if a person's salary is greater than $90,000, and only $2,000 otherwise:

```
float salary;
// Retrieve the salary of the person from some data source here
int bonus = salary>90000?3000:2000;
```

### The switch Statement

The `switch` statement is used when the program execution is defined based on multiple selections. Even though execution can be defined with a long `if-else if` construct, the `switch` statement can be more readable. In the following code fragment the bonus amount is selected based on the salary band of the worker. Imagine that this code is written in a loop that reads data from a database of employees of some firm, and that the `salaryBand` value is assigned right before the `switch` statement.

```
        int salaryBand;
        int bonus;
        // Retrieve the salary band of the person from some data source here

              switch(salaryBand){
               case 1:
                     bonus=1000;
                     break;
               case 2:
                     bonus=2000;
                     break;
               case 3:
                     bonus=6000;
                     break;
               case 4:
                     bonus=10000;
                     break;
               default:
```

```
                    // wrong salary band
                    System.out.println("Invalid salary band");
                }
```

At the time of this writing Java doesn't allow the use of `String` as a `switch` variable — only integer or enumerated constants are allowed. There is a proposal to support strings in Java 7, but it's not clear if this support will make it into production release.

### Command-Line Arguments

In Lesson 1 you learned how to start a Java program from a command line. After development in Eclipse or another IDE is done, Java programs are **deployed in production** and will be started from a command line — usually you get an icon to click that runs a command to start a program, but under the hood the operating system executes a command that starts your program. You can't expect an accountant to have Eclipse installed to run a tax calculation program, right? For example, to run the TestTax program from the command window you need to open a command window, change the directory to `Lesson 4` and enter the following:

```
java TestTax
```

You can run only those Java classes that have the method `main`, which takes a `String` array as an argument. This array is the means of passing some data from the command line to your program during start-up. For example, you can pass gross income, state, and number of dependents to `TestTax` by starting it from the command line as follows:

```
java TestTax 50000 NJ 2
```

The method `main(String[] args)` of the class `TestTax` receives this data as a `String` array that I decided to call `args`. This array will be automatically created by JVM, and will be large enough to accommodate all the argu-

ments entered from the command line. This array will be populated as follows:

```
args[0] = "50000";
args[1] = "NJ";
args[2] = "2";
```

Command-line arguments are always being passed to a program as `String` arrays. It's the responsibility of the programmer to convert the data to the appropriate data type. The wrapper Java classes that were introduced in Lesson 3 can come in handy. For example, the wrapper class `Double` has the function `parseDouble` to convert `String` to `double`:

```
double grossIncome = Double.parseDouble(args[0]);
```

Review the code of the TestTax program from Listing 4-6, which has hard-coded values of gross income, state, and number of dependents. By using command-line arguments you can make the program more generic and useful not only for people from New Jersey having two dependents and an annual income of $50,000. You'll work on this assignment in the Try It section.

### Try It

Your assignment is to introduce command-line arguments into the program from Listing 4-6.

### 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 Lesson5 folder in the download.

### Step-by-Step

**1.** In Eclipse IDE, copy the `TestTax` class from the Lesson4 project to Lesson5. Also copy the `Tax` class from the project Lesson3 that has an `if` statement in the `calcTax()` method.

**2.** Remove the three lines that initialize `grossIncome`, `state` and `dependents` with hard-coded values.

**3.** Add the following code fragment to ensure that the program has been started with three arguments. If it has not, print the error message and exit the program.

```
if (args.length != 3){
      System.out.println("Sample usage of the program:" +
                         "  java TestTax 50000 NJ 2");
      System.exit(0);
}
```

**4.** Add the following statements to the method `main()` to get the values passed from the command line, convert them to appropriate data types, and initialize the variables `grossIncome`, `state`, and `dependents`:

```
   double grossIncome = Double.parseDouble(args[0]);
   String state = args[1];
   int dependents = Integer.parseInt(args[2]);
```

**5.** Right-click the Lesson5 project in Eclipse IDE and select the Properties menu item (note the location of the compiled classes of


Stringray Group Inc. et al v. Hernandez-Mondragon

your project — in my case it was `c:\practicalJava\workspase\Lesson5\bin`).

**6.** Open the command window and change the directory to the one that contains the file `TestTax.class`.

**7.** Run your program several times, specifying different values for the command line arguments. The program should print different values for the calculated tax.
To get the sample database files you can download the content of the Lesson 5 folder from the book's website at **www.wrox.com**.

---



Please select Lesson 5 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 6

# Packages, Interfaces, and Encapsulation

If you want to find out if a particular programming language is object-oriented, ask if it supports **inheritance**, **encapsulation**, and **polymorphism**. You know by now that Java supports inheritance, which lets you design a class by deriving it from an existing one. This feature allows you to reuse existing code without copy-pasting reusable code fragments from other classes — the class `NJTax` from Listing 3-5 was designed this way.

In Lessons 6 and 7 you learn what encapsulation means and continue studying coding techniques and best practices. While this lesson will show you several short code examples illustrating certain programming topics, the next one will bring all the pieces you've learned so far together in one larger practical application.

### Java Packages

A decently size project can have hundreds of Java classes and you need to organize them in **packages** (think file directories). This will allow you to categorize files, control access to your classes (see the section "Access Levels" later in this chapter), and avoid potential naming conflicts: If both you and your colleague coincidentally decide to call a class `Util`, this won't be a problem as long as these classes are located in different packages.

Sometimes you'll be using third-party libraries of Java classes written in a different department or even outside your firm. To minimize the chances that package names will be the same, it's common to use so-called **reverse domain name** conventions. For example, if you work for a company called Acme, which has the website **acme.com**, you can prefix all package names with `com.acme`. To place a Java class in a certain package, add the package statement at the beginning of the class (it must be the first non-comment statement in the class). For example, if the class `Tax` has been developed for the Accounting department, you can declare it as follows:

```
package com.acme.accounting;
class Tax {
  // the class body goes here
}
```

If you declare the class `Tax` as shown in the preceding code, the file `Tax.java` must be stored in the corresponding directory tree:

```
Com
     Acme
        Accounting
```

Java classes are also organized into packages and the fully qualified name of a class consists of the package name followed by the class name. For example, the full name of the Java class `Double` is `java.lang.Double`, where `java.lang` is the package name. As a matter of fact, `java.lang` is the only package name that you don't have to explicitly mention in your code in order for its classes to be found, unless all classes are located in the same package.

The program documentation on all Java 6 packages is available at the following URL: **http://download.oracle.com/javase/6/docs/api/**.

Let's say your `Tax` class needs to connect to some URL on the Net with the help of the class `URL` located in the `java.net` package. You can write code containing the fully qualified name of this second class:

```
java.net.URL myURL = new java.net.URL ("http://www.acme.com");
```

But instead of using this rather long notation, include the `import` statement right above the class declaration, and then use just the name of the class:

```
import java.net.URL;
class Tax{
   URL myURL = new URL("http://www.acme.com");
   ...
}
```

If you need to import several classes from the same package, use the wild card in the `import` statement rather then listing each of the classes on a separate line:

```
import java.net.*;
```

### Encapsulation

Encapsulation is the ability to hide and protect data stored in Java objects. You may ask, “Who are the bad guys who want to illegally access my data?” It’s not about bad guys. When a developer creates a Java class, he or she plans for a certain use pattern of this code by other classes. For example, the variable `grossIncome` should not be modified directly, but via a method that performs some validation procedures to ensure that the value to be assigned meets application-specific rules.

A Java developer may decide to “hide” 15 out of 20 variables, say, so other classes can’t access them. Imagine how many parts exist in a car and how many functions those parts can perform. Does the driver need to know about all of them? Of course not. The driver needs to know how to start and stop the car, how to indicate turns, open the windows, turn on the wipers, and do a few dozen other simple operations, which in programming jargon can be called the car’s **public interface**.

### Access Levels

Java classes, methods, and member variables can have `public`, `private`, `protected`, and `package` access levels, for example:

```
public class Tax {
   private double grossIncome;
   private String state;
   private int dependents;
   protected double calcTax(...) {...}
}
```

The keyword `public` means that this element (a class, a method, or a variable) can be accessed from any other Java class. The keyword `protected` makes the element “visible” not only from the current class, but from its subclasses too.

The keyword `private` is the most restrictive one, as it makes a member variable or a method accessible only inside this class. For example, our class `Tax` may need some additional internal methods that could be called from the method `calcTax()`. The users of the class `Tax` do not need to know about these methods and they should be declared as `private`.

If you do not specify any access level, the default is `package`, which means that only classes located in the same package will have access to this method or variable. Java classes should expose only the methods that outsiders have to know, such as `calcTax()`.

If you are not sure which access level to give to methods or variables, just make them all `private`; if later on during development some other class needs to access them, you can always change the access level to be less restrictive. This will protect all the internals of your application from misuse. Think of it this way: "I want to sell my class `Tax` to various accounting firms across the country. If their software developers integrate my class with their existing systems, what are the methods that they must know about to be able to calculate tax?" If car designers did not ask themselves similar questions, drivers would need to press dozens of buttons just to start the engine.

### The Keyword final

The keyword `final` can have different meanings depending on the context. It's explained in the next sections.

### final Variables

You can use the keyword `final` while declaring variables, methods, and classes. A `final` variable becomes a constant (see Lesson 3) that can be initialized only once and can't change its value during the run time. Some people may argue that a constant and a variable that can get initialized only once are not the same thing, but the fact that you can't change their values makes them very similar.

Even though you can declare constants inside a method, it's more common to declare them on the class level so they can be reused by several methods of the same class. You can also declare a variable with the keyword `static` and with a proper access method so they can be used:

```
static final int BOILING_TEMP = 212; //in Fahrenheit
```

### final Methods

If you declare a class method with a `final` keyword, this method can't be overridden if someone decides to extend the class. Today it may seem obvious to you that a particular method will never ever need to be overridden. What are the chances that the formula to convert Fahrenheit to Celsius will be changed any time soon?

```
static final double convertToCelsius(double far){
           return ((far - 32) * 5 / 9);
}
```

But in many cases developers create reusable libraries of Java classes, finalizing functionality that's not written in stone. While it may seem to you that a particular method will never need to be overridden, you might have predicted all use patterns of this class. If this happens, some other developer will have to jump through hoops to create another version of your method in a subclass.

Many years ago the Java compiler optimized (inlined) final methods. It doesn't do that anymore — all methods are optimized by the Hotspot JVM. Just think twice before making a method `final`.

### final Classes

If you declare a class as `final`, no one will be able to subclass it. For example, Java's class `String` has been created as immutable and therefore declared as `final` (see http://java.sun.com/javase/6/docs/api/java/lang/String.html). If a class is declared as `final`, all its methods become implicitly `final`.

### Interfaces

There are different approaches to how to start designing a class. Some people think of the behavior it should support. The behavior is implemented in a class in the form of methods, which can be declared in a separate entity called an **interface.** Then have your class implement this interface. **Listing 6-1** shows an interface with one method declaration.

In general, you should have a reason for declaring the methods separately from the class that will implement it, and you'll see these reasons in the next lesson when you learn about polymorphism, For now, let's just get familiar with the syntax of defining and using interfaces. The interface will not have any implementations — just declarations. (There is an exception: Marker interfaces don't even have declarations. The next section is devoted to this type of interface.)

When a class declares that it implements a certain interface, it guarantees that it contains implementation for all methods from this interface. And a class can implement more than one interface: just separate their names with commas.

Let's say you have two types of workers in your organization, employees and contractors, and that you create the classes `Employee` and `Contractor` to implement functionalities that reflect specifics of these different groups. Each person is entitled to a pay raise, though for employees

this means a salary increase and for contractors it's an increase of an hourly or daily rate.

Instead of creating two different methods in these classes, it's better to define an interface called, say, `Payable` that contains the declaration of the method `increasePay()`, and to have both classes implement it, as in Listings 6-1, 6-2, and 6-3. Every method declared in the interface automatically becomes `public`.




**Listing 6-1:** Payable interface

```
public interface Payable {

       boolean increasePay(int percent);

}
```




**Listing 6-2:** Class Employee

```
public class Employee implements Payable {

       public boolean increasePay(int percent) {
              // implement salary raise here
       }

}
```




**Listing 6-3:** Class Contractor

```
public class Contractor implements Payable {
       public boolean increasePay(int percent) {
              // implement hourly rate increase here
       }
}
```

Since both `Employee` and `Contractor` contain the clause `implements Payable`, you must implement the `increasePay()` method in each of the classes, or your code won't compile. Creating classes with common interfaces leads to cleaner design of your application and makes the code more readable. But what's more important is that with the help of interfaces you can introduce polymorphic behavior to your program, and this will be illustrated in Lesson 7.

Besides method signatures, Java interfaces can contain declarations of `final` variables. For example, you can create a `final` variable in the `Payable` interface for the maximum percentage of pay increase (all variables declared in the interface automatically become `public static final`).

```
int INCREASE_CAP = 20;
```

Since both the `Employee` and `Contractor` classes implement `Payable`, they can both include `if` statements in the implementation of `increasePay()` to ensure that the provided percentage increase is less than `INCREASE_CAP`. If in the future the cap changes, you need to change it in only one place — the `Payable` interface.

Some software developers create Java interfaces that contain only `final` variables storing important application constants. Implementing such interfaces will make these constants available in the class that implements the interface(s). Not everyone approves such usage of interfaces, as it can create a messy situation when a class that implements interfaces with static constants exposes a new set of public APIs (those `final` variables) rather than just using these values internally. You can read more about this anti-pattern at the following URL: **http://java.sun.com/j2se/1.5.0/docs/guide/language/static-import.html**.

### Marker Interfaces

**Marker interfaces** are those that don't have any methods declared. One example of such an interface is `Serializable`, covered in Lesson 17. You don't need to write any implementation of these interfaces; the Java compiler will take care of this for you. Objects of a class that implements a marker interface will support a certain functionality. For example, if a class implements `Serializable`, JVM will be able to **serialize** it — turn into a string of bytes (in the server's JVM) in such a way that the string can be sent to another JVM (on the client's machine), which will be able to recreate the instance of the object, or **de-serialize** it.

Also, Java has an operator `instanceof` (see next section) that can check during the run time if an object is of a certain type. You can use the operator `instanceof` as shown below to check if the object implements a marker interface, or any other type of interface for that matter:

```
if (receivedFromServerObj instancef Serializable) {
    // do something
}
```

### Casting

All Java classes form an inheritance tree with the class `Object` on top of the hierarchy — all Java classes are direct or indirect descendants of `Object`. When you declare a non-primitive variable, you are allowed to use either the exact data type of this variable or one of its ancestor data types. For example, if the class `NJTax` extends `Tax`, each of the following lines is correct.

```
NJTax myTax1   = new NJTax();
Tax myTax2     = new NJTax();  // upcasting
Object myTax3  = new NJTax();  // upcasting
```

Java is smart enough to automatically cast an instance of the class to its ancestor. When the variable has more generic type than an instance of the object this is called **upcasting**. Let's say the class object has 10 methods and class variables defined, the class `Tax` (an implicit subclass of `Object`) adds five more methods and variables (making 15), and `NJTax` adds another two (total 17). The variable `myTax1` will have access to all 17 methods and variables, `myTax2` will see only 15, and `myTax3` just 10. Why not always use exact types in variable declarations?

Let's say you need to write a program that will process data about workers of a certain company. Some of them are full-time employees and some are contractors, but you'd like to read their data from some data source and into the same array. Arrays can store only objects of the same type, remember?

Since Java can automatically upcast the objects, you can create a class called `Person` with two subclasses, `Employee` and `Contractor`, and then read the records from a database. Based on the employment type you can then create an appropriate object instance and put it into an array of type `Person`:

```
Person workers[] = new Person [100];
workers[0] = new Employee("Yakov", "Fain");
workers[1] = new Employee("Mary", "Lou");
workers[2] = new Contractor("Bill", "Shaw");
...
```

Of course, you could've created two separate arrays, one for employees and one for contractors, but I'm laying the foundation here for explaining polymorphism — a powerful concept in object-oriented languages. You'll see a concrete example of polymorphism in Lesson 7.

At some point you'll need to process the data from the array `workers`. In a loop you can test the data type of the current element of the array with the operator `instanceof`, and then downcast the object (it can't be done automatically) to `Employee` or `Contractor` and process it accordingly.

```
for (int i; i<20; i++){
     Employee currentEmployee;
     Contractor currentContractor;

     if (workers[i] instanceof Employee){
          currentEmployee = (Employee) workers[i];
          // do some employee processing here
     } else if  (workers[i] instanceof Contractor){
         currentContractor = (Contractor) workers[i];
         // do some contractor processing here
     }
}
```

Placing a data type in parentheses in front of another object means that you want to **downcast** this object to the specified type. You can downcast an object only to one of its descendant data types. Even though the preceding code has correct syntax, it doesn't represent the

best practice for processing similar objects. In the next lesson you'll see how to use polymorphism to do it in a more generic way.

If a class implements an interface, you can cast its instance to this interface. Say that a class called `Employee` implements `Payable`, `Insurable`, and `Pensionable` interfaces:

```
class Employee implements Payable, Insurable, Pensionable {
// implementation of all interfaces goes here
}
```

If you are interested in the `Insurable` behavior of your code, there's no need to cast the `Object` to type `Employee` — just cast it to the `Insurable` type as shown in the following code fragment. Keep in mind though that if you do so, the variable `currentEmployee` will expose access only to those methods that were declared in the `Insurable` interface.

```
Insurable currentEmployee;

if (workers[i] instanceof Insurable){
  currentEmployee = (Insurable) workers[i];
  // do some insurance-specific processing here
}
```

### Try It

The goal of this assignment is to start using packages, protect data using private variables and define first interfaces. You create a simple program that will increase pay, which will be implemented differently for employees and contractors. After completing this assignment you'll

have working but not perfect code. What can be improved will be explained in Lesson 7.

### 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 Lesson6 folder in the download.

---

### Step-by-Step

**1.** In the Eclipse IDE, create a new project called Lesson6.

**2.** Create the `Payable` interface as per **Listing 6-1** in the package `com.practicaljava.lesson6` — you can enter the name of the package in the Eclipse New Java Class window. Declare a `final` variable there:

```
int INCREASE_CAP = 20;
```

**3.** Create a class called `Person`:

```
package com.practicaljava.lesson6;

public class Person {
	private String name;

	public Person(String name){
		this.name=name;
	}

	public String getName(){
		return "Person's name is " + name;
	}
```

```
}
```

**4.** Create the classes `Employee` and `Contractor` in the package `com.practicaljava.lesson6`. Each class should extend `Person` and implement `Payable`. Use the Eclipse button Add to automatically include the `Payable` interface in declarations of the classes `Employee` and `Contractor`.

**5.** Check your file system to ensure that the files were created in your workspace in the directory `com/practicaljava/lesson6`.

**6.** Create a class called `TestPayIncrease` with a method called `main()`. Don't specify any package — this class will be created in a different directory.

**7.** Try to create an instance of the class `Employee` in the method `main()` of `TestPayIncrease`:

```
Employee myEmployee = new Employee();
```

You'll get an error: `Employee` can't be resolved to a type. No wonder — it's located in a different package. Move the mouse over `Employee` and Eclipse will offer you a fix — add an `import` statement:

```
import com.practicaljava.lesson6.Employee;
```

Select this fix and later add the `import` statement for all required classes.

**8.** In the `main()` method of the class `TestPayIncrease` create an array of employees and contractors and call the function `increasePay()` for each element of the array:

```
public static void main(String[] args) {

        Person workers[] = new Person[3];
```

```
            workers[0] = new Employee("John");
            workers[1] = new Contractor("Mary");
            workers[2] = new Employee("Steve");

                  Employee currentEmployee;
                  Contractor currentContractor;

             for (Person p: workers){
                 if (p instanceof Employee){
                      currentEmployee = (Employee) p;
                      currentEmployee.increasePay(30);

                 }else if  (p instanceof Contractor){
                     currentContractor = (Contractor) p;
                     currentContractor.increasePay(30);
                 }
            }
}
```

**9.** Implement the `increasePay()` method in `Employee` — don't put any restrictions on pay increases. Here's the body of `increasePay()`:

```
       System.out.println("Increasing salary by " + percent + "%. "+ getName());
       return true;
```

**10.** Implement the `increasePay()` method in the class `Contractor`. If the percentage of the increase is below `INCREASE_CAP`, print a message similar to the one in the preceding code. Otherwise, print a message explaining that you can't increase a contractor's rate by more than 20 percent.

**11.** Run the `TestPayIncrease` program. It should produce output similar to the following:

```
Increasing salary by 30%. Person's name is John
Sorry, can't increase hourly rate by more than 20%. Person's name is Mary
Increasing salary by 30%. Person's name is Steve
```

To get the sample database files you can download Lesson 6 from the book's website at **www.wrox.com**.



Please select Lesson 6 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 7

# Programming with Abstract Classes and Interfaces

In this lesson you learn about so-called abstract classes, and then you build a complete application that will illustrate how to design and implement programs with abstract classes and interfaces. You also learn about the notion of **polymorphism**.

### Abstract Classes

If a class is declared **abstract** it can't be instantiated. The keyword `abstract` has to be placed in the declaration of a class. The abstract class may have abstract method(s). The question is "Who needs a class that can't be instantiated?"

It's easiest to answer this question by showing you how to use abstract classes while designing an application. While previous lessons ended with assignments, this lesson will start with one.

### Assignment

A company has employees and contractors. Design the classes without using interfaces to represent the people who work for this company. The classes should have the following methods:

```
changeAddress()
promote()
giveDayOff()
increasePay()
```

A one-time promotion means giving one day off and raising the salary by a specified percentage. The method `increasePay()` should raise the yearly salary for employees and increase the hourly rate for contractors.

### Solution with an Abstract Class

Since `increasePay()` has to be implemented differently for `Employee` and `Contractor`, let's declare it as abstract. The class `Person` will also have three concrete methods. The fact that the abstract class `Person` cannot be instantiated forces us, the developers, to implement abstract methods in subclasses.

Let's start by redesigning the class `Person` from the Try It section of Lesson 6. That version of the class didn't have the method `increasePay()`, which was a part of the `Payable` interface. As per the previous lesson's assignment, add the following concrete methods: `changeAddress()`, `giveDayOff()`, and `promote()` (see **Listing 7-1**).

This is a different approach from the one in Lesson 6, which used interfaces — here some methods are implemented in the superclass and some are not. This solution won't be using any interfaces.




**Listing 7-1:** Abstract class Person

```
package com.practicaljava.lesson7;

public abstract public class Person {

	private String name;

	int INCREASE_CAP = 20;  // cap on pay increase

	public Person(String name){
		this.name=name;
	}

	public String getName(){
		return "Person's name is " + name;
	}

	public void changeAddress(String address){
		   System.out.println("New address is" + address);
	}

	private void giveDayOff(){
		System.out.println("Giving a day off to " + name);
	}

	public void promote(int percent){
		   System.out.println(" Promoting a worker...");
		   giveDayOff();

		   //calling an abstract method
		   increasePay(percent);
	 }

	 // an abstract method to be implemented in subclasses
	 public abstract boolean increasePay(int percent);
}
```

Please note that even though the method `increasePay()` is abstract, the designer of the class `Person` doesn't have to know the specifics of raising pay, and subclasses may even be programmed by other developers. But he or she can write code that calls `increasePay()`, as in the method `promote()`. This is allowed because by the time the concrete class is instantiated, this method will definitely have been implemented. For simplic-

ity, I didn't write any code that looks like actual increasing pay — this is irrelevant for understanding the concept of abstract classes.

The next step is to create the subclasses `Employee` and `Contractor`, implementing the method `increasePay()` in two different ways, as shown in in Listings 7-2 and 7-3.




**Listing 7-2:** Class Employee

```
package com.practicaljava.lesson7;

public class Employee extends Person{

	public Employee(String name){
		super(name);
	}
	public boolean increasePay(int percent) {
		System.out.println("Increasing salary by " +
			percent + "%. "+ getName());
		return true;
	}
}
```




**Listing 7-3:** Class Contractor

```
package com.practicaljava.lesson7;

public class Contractor extends Person {

	public Contractor(String name){
		super(name);
```

```
        }
        public boolean increasePay(int percent) {
               if(percent < INCREASE_CAP){
                  System.out.println("Increasing hourly rate by " +
                                      percent + "%. "+ getName());
                  return true;
               } else {
                  System.out.println("Sorry, can't increase hourly rate by more
                         than " + INCREASE_CAP + "%. "+ getName());
                  return false;
               }
         }
}
```

Programmers writing subclasses are forced to write an implementation of `increasePay()` according to its signature, declared in the abstract class. If they declare a method increasing pay that has a different name or argument list, their classes will remain abstract. So they don't have a choice — they have to play by the rules dictated in the abstract class.

The class `TestPayIncrease2` in **Listing 7-4** shows how to use the classes `Employee` and `Contractor` for promoting workers.




**Listing 7-4:** Class TestPayIncrease2

```
import com.practicaljava.lesson7.*;

public class TestPayIncrease2 {

      public static void main(String[] args) {

        Person workers[] = new Person[3];
              workers[0] = new Employee("John");
              workers[1] = new Contractor("Mary");
              workers[2] = new Employee("Steve");
```

```
                for (Person p: workers){
                        p.promote(30);
                }
        }
}
```

Compare the code of `TestPayIncrease2` to `TestPayIncrease` from the Try It section of Lesson 6. Which one do you like better? I like `TestPayIncrease2`, which exhibits polymorphic behavior, explained next.

### Polymorphism

Polymorphism is easier to understand through an example. Let's look at the classes `Person`, `Employee`, and `Contractor` from a different angle. The code in **Listing 7-4** populates an array, mixing up the instances of the classes `Employee` and `Contractor` with hard-coded names. In real life the data about workers usually comes from an external data source. For example, a program could get a person's work status from the database and instantiate an appropriate concrete class. The class `TestPayIncrease2` gives an additional vacation day and attempts to increase the salary or hourly rate of every worker by 30 percent.

Note that even though the loop variable `p` is of its ancestor's type `Person` in **Listing 7-4**, at every iteration it actually points at either an `Employee` or a `Contractor` instance. The actual object type will be evaluated only during the run time. This feature of object-oriented languages is called **run-time binding** or **late binding**.

The output of the class `TestPayIncrease2` will look as follows:

```
Promoting a worker...
Giving a day off to John
Increasing salary by 30%. Person's name is John
 Promoting a worker...
```

```
Giving a day off to Mary
Sorry, can't increase hourly rate by more than 20%. Person's name is Mary
 Promoting a worker...
Giving a day off to Steve
Increasing salary by 30%. Person's name is Steve
```

Both classes, `Employee` and `Contractor`, were inherited from the same base class, `Person`. Instead of having different methods for increasing the worker's compensation based on the worker's type, we give a polymorphic behavior to the method `increasePay()`, which applies different business logic depending on the type of the object.

Although it looks as if we're calling the same method, `promote()`, on every object from the array `workers`, this is not the case. Since the actual object type is evaluated during run time, the pay is raised properly according to this particular object's implementation of the method `increasePay()`. This is polymorphism in action.

The `for` loop in the class `TestPayIncrease2` will remain the same even if we add some other types of workers inherited from the class `Person`. For example, to add a new category of worker — a foreign contractor — we'll have to create a class called `ForeignContractor` derived from the class `Person` and implement the method `increasePay()` there. The class `TestPayIncrease2` will keep evaluating the actual type of `Person`'s descendants during run time and call the proper implementation of the method `increasePay()`.

Polymorphism allows you to avoid using `switch` or `if` statements with the expensive operator `instanceof`, which you used in Lesson 6. Would you agree that even though `TestPayIncrease` is producing the same results, its code looks pretty ugly compared to `TestPayIncrease2`? The code in `TestPayIncrease` works more slowly than the polymorphic version, and its `if` statement has to be modified every time a new type of worker is added.

### Making the Interface Solution Polymorphic

Could you create the polymorphic solution in the example with interfaces from Lesson 6? Yes, you could. Even though the array `workers` has been declared of type `Person`, it is populated by the objects that implement the `Payable` interface.

Instead of casting the objects from this array to either `Employee` or `Contractor`, you would cast these objects to `Payable` and call the `increasePay()` method without worrying too much about whether the current worker is an employee or a contractor (see **Listing 7-5**).




**Listing 7-5:** Class TestPayIncreasePoly

```
import com.practicaljava.lesson6.*;

public class TestPayInceasePoly {

        public static void main(String[] args) {

        Person workers[] = new Person[3];
              workers[0] = new Employee("John");
              workers[1] = new Contractor("Mary");
              workers[2] = new Employee("Steve");

              for (Person p: workers){
                 ((Payable)p).increasePay(30);
              }
        }
}
```

Note that the variable `p` is cast to `Payable` first, and only after the method `increasePay()` can be called. To enforce the sequence of these operations this code has been surrounded with parentheses. The Eclipse IDE comes in handy with its context-sensitive help — when you type the period after the parentheses it readily shows you `increasePay()` in the list

of available methods. I always use this little help: If a method name I'm about to type is not shown there, I must be doing something wrong with the types; otherwise Eclipse would have helped me out.

The other solution that would eliminate the need of casting to `Payable` is having the class `Person` implement `Payable`. In this case the loop from **Listing 7-5** would look as follows:

```
for (Payable p: workers){
   increasePay(30);
}
```

What can go wrong during the execution of the code from **Listing 7-5**? What if a developer creates a class called `ForeignContractor` without implementing the `Payable` interface, and this error makes it into the array `workers`? You'll get a run-time casting error — JVM can't cast an object to `Payable` if it doesn't implement the `Payable` interface. You'll have a chance to see this error for yourself in the Try It section.

### Interfaces versus Abstract Classes

The next question is when should you use interfaces and when should you use abstract classes. If two or more classes have lots of common functionality, but some methods should be implemented differently, you can create a common abstract ancestor and as many subclasses inheriting this common behavior as needed. Declare in the superclass as abstract those methods that subclasses should implement differently, and implement these methods in subclasses.

If several classes don't have common functionality but need to exhibit some common behavior, do not create a common ancestor, but have them implement an interface that declares the required behavior. This scenario was not presented in the "Interfaces" section of Lesson 6, but

it's going to be a part of the hands-on exercise in the Try It section of this lesson.

Interfaces and abstract classes are similar in that they ensure that required methods will be implemented according to required method signatures. But they differ in how the program is designed. While abstract classes require you to provide a common ancestor for the classes, interfaces don't.

Interfaces could be your only option if a class already has an ancestor that cannot be changed. Java doesn't support multiple inheritance — a class can have only one ancestor. For example, to write Java applets you must inherit your class from the class `Applet`, or in the case of Swing applets, from `JApplet`. Here using your own abstract ancestor is not an option.

While using abstract classes, interfaces, and polymorphism is not a must, it certainly improves the design of Java code by making it more readable and understandable to others who may need to work on programs written by you.

### Try It

In the first part of the assignment your goal is to break the code from **Listing 7-5** to produce the run-time error `ClassCastException`. In the second part of the assignment you'll need to rewrite the assignment from Lesson 6 to keep the `Payable` interface but remove the common ancestor `Person`.

### 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 Lesson7 folder in the download.

---

### Step-by-Step

#### *Part 1*

**1.** In Eclipse open the project Lesson6 — yes, the one from the previous lesson.

**2.** Create a new class called `ForeignContractor`, as shown in the following code. Note that this class doesn't implement the `Payable` interface:

```
package com.practicaljava.lesson6;

public class ForeignContractor extends Person {

	public ForeignContractor(String name){
		super(name);
	}
	public boolean increasePay(int percent) {

		System.out.println("I'm just a foreign worker");
		return true;
	}
}
```

**3.** Create the class `TestPayIncreasePolyError`, adding an instance of the `ForeignContractor` class:

```
import com.practicaljava.lesson6.*;

public class TestPayIncreasePolyError {

   public static void main(String[] args) {

       Person workers[] = new Person[3];
```

```
        workers[0] = new Employee("John");
        workers[1] = new Contractor("Mary");
        workers[2] = new ForeignContractor("Boris");


               for (Person p: workers){
                        ((Payable)p).increasePay(30);
               }
        }
}
```

**4.** Run the program `TestPayIncreasePolyError`. Observe the output in the console view — you'll get the run-time error `java.lang.ClassCastException` on the third element of the array. Note the number `14` — this is the line number of `TestPayIncreasePolyError` program, which casts each object to the `Payable` interface.

```
Increasing salary by 30%. Person's name is John
Sorry, can't increase hourly rate by more than 20%. Person's name is Mary
Exception in thread "main" java.lang.ClassCastException:
com.practicaljava.lesson6.ForeignContractor cannot be cast to
com.practicaljava.lesson6.Payable
         at TestPayInceasePolyError.main(TestPayInceasePolyError.java:14)
```

**5.** Modify the code of `TestPayIncreasePolyError`, changing the type of the array from `Person` to `Payable` and changing the type of the loop variable accordingly:

```
Payable workers[] = new Payable [3];
workers[0] = new Employee("John");
workers[1] = new Contractor("Mary");
workers[2] = new ForeignContractor("Boris");

       for (Payable p: workers){
                ((Payable)p).increasePay(30);
       }
```

**6.** Observe that now you are getting a Java compiler error preventing you from even adding to the array the instance of `ForeignContractor` because it doesn't implement `Payable`. Predicting and preventing run-time

errors is a very important task for every software developer, and this subject will be covered in detail in Lesson 13.

***Part 2***

**1.** In the project Lesson7 select File⇒ New create the new package `com.practicaljava.lesson7.tryit`.

**2.** In Eclipse copy the `Payable` interface from Lesson6 to the package `com.practicaljava.lesson7.tryit`.

**3.** In the same package create the class `Employee` as follows:

```
package com.practicaljava.lesson7.tryit;

public class Employee implements Payable{
	private String name;

	public Employee(String name){
		this.name=name;
	}

	public boolean increasePay(int percent) {
		System.out.println("Increasing salary by " + percent
		+ "%: " + name);
		return true;
	}
}
```

**4.** In the same package create the class `Contractor` as follows:

```
package com.practicaljava.lesson7.tryit;

public class Contractor implements Payable {

	private String name;

	public Contractor(String name){
		this.name=name;
	}
	public boolean increasePay(int percent) {
```

```
            if(percent < Payable.INCREASE_CAP){
                   System.out.println("Increasing hourly rate by " +
                        percent + "%. ");
                   return true;
             } else {
                   System.out.println("Sorry, can't increase hourly rate by
                      more than " + Payable.INCREASE_CAP + "%: " + name);
                   return false;
             }
```

Note that neither `Employee` nor `Contractor` extends `Person` any longer. Both classes are free to extend any other classes now, but on the other hand each of them has to declare the variable name and the method `getName()`, which was done once in the class `Person` before.

**5.** Create a class called `TestPayIncreaseInterface`:

```
import com.practicaljava.lesson7.tryit.*;

public class TestPayIncreaseInterface {

      public static void main(String[] args) {

        Payable workers[] = new Payable [3];
              workers[0] = new Employee("John");
              workers[1] = new Contractor("Mary");
              workers[2] = new Employee("Steve");

                   for (Payable p: workers){
                           ((Payable)p).increasePay(30);
                   }
             }
}
```

**6.** Run this program — it should produce the following output:

```
Increasing salary by 30%: John
Sorry, can't increase hourly rate by more than 20%: Mary
Increasing salary by 30%: Steve
```

To get the sample database files you can download Lesson 7 from the book's website at **www.wrox.com**.



Please select Lesson 7 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 8

# Introducing the Graphic User Interface

These days people are accustomed to working with applications that have rich user interfaces. If you develop rich Internet applications, JavaFX is the newest platform for development offered by Oracle. But Java itself offers a library of components called Swing, which is used for building graphic user interfaces (GUIs) for desktop applications, as well as a web program called applets (see Lesson 10). In this and the following lesson you'll learn the principles of building GUIs while developing a simple desktop calculator using the Java Swing library.



Eclipse Foundation offers another library of UI components called SWT, which is out of the scope of this book.

### Swing Basics

Originally Java offered a pretty basic library of UI-related classes called the Abstract Windowing Toolkit (AWT). A couple of years later a new widget toolkit called Swing was introduced. It offers a lighter set of UI components while keeping the main idea intact — to keep UI development independent of the specifics of the end user's operating system. Today developers are trying to create UIs that appear to be native to the OS, whether that is Windows, Mac OS, iOS, or Android.

Eventually the market share of Swing-based UIs will diminish, but at the time of this writing it's still widely used by enterprises, and skilled Swing developers remain in big demand.

Swing offers you everything you need to build UIs in Java: There are controls to represent buttons, drop-downs, grids, scrollbars, trees, tab folders, and so on. Typically you'll be creating UIs by combining controls into containers (such as `JPanel`) that support various layouts that enable controls to be arranged as you or a graphic designer envision. In this lesson you use some of the Swing components while creating a UI for a simple calculator.

A complete discussion of the Swing library is out of the scope of this book, but there are plenty of books and technical articles covering this subject. The official online Swing tutorial is located at **http://java.sun.com/docs/books/tutorial/uiswing**.

Swing classes are located in the `javax.swing` package, and the process of creating a UI comes down to extending some of these classes to display the UI and respond to various user- and system-generated events. You create a top-level window with a title and border by instantiating the class `JFrame`, as in **Listing 8-1**.




**Listing 8-1:** An empty descendant of JFrame

```
import javax.swing.JFrame;

public class HelloWorld extends JFrame {

    public HelloWorld(){
       setSize(200,300);
       setTitle("Hello World");
       setVisible(true);
```

```
    }
        public static void main(String[] args) {
        HelloWorld myHello = new HelloWorld();
        }

}
```

The constructor of this version of Hello World sets the size and title of the window and makes it visible. Run this program and it'll show a small empty window, as in **Figure 8-1**.

**Figure 8-1**




`JFrame` is an example of a container that can hold UI controls, which must be instantiated first and then added to the container. For example, to add a button you can include the following two lines to the constructor in **Listing 8-1**:

```
        JButton myButton = new JButton ("Click me");
        add(myButton);
```

This code will work, but it doesn't specify the size of the button, where to put it, or whether there should be space between the components and the borders. Without layout instructions, the entire empty space in **Figure 8-1** will be occupied by one huge button.

Usually `JFrame` includes some other containers, where you'll place controls such as `JButton`, `JTable`, and `JList`. Java Swing comes with **layout managers** that help you arrange all these controls appropriately. A sample coding process for creating a `JFrame` containing `JPanel` goes like this:

**1.** Create a `JPanel`.

**2.** Assign a layout manager to it.

**3.** Instantiate some Swing controls and add them to the panel.

**4.** Add the panel to the top-level container — `JFrame` — by calling the `setContentPane()` method.

**5.** Set the frame's size and make it visible.

You can assign different layout managers to your containers to create pretty sophisticated windows. But displaying a window with properly laid-out components is only half of the job, because these controls should know how to respond to various **events**, such as a click on a button. This lesson covers the basics of displaying UI components; the next one is about writing code for responding to events.

### Layout Managers

The simplest layout manager is `FlowLayout`, which allocates all components being added to the container horizontally. When there's no room for the next component, `FlowLayout` uses the next row, and the process repeats.

### A Simple Calculator with FlowLayout

The best way to learn layout management is by trying to use it in practice. Let's create a UI for a simple calculator that can accept two numbers and display the result. Create a new Eclipse project called Lesson8 and a new class called `SimpleCalculator` with the following code:




**Listing 8-2:** Calculator with FlowLayout

```
import javax.swing.*;
import java.awt.FlowLayout;

public class SimpleCalculator {
 public static void main(String[] args) {
  // Create a panel
       JPanel windowContent= new JPanel();

  // Set a layout manager for this panel
       FlowLayout fl = new FlowLayout();
       windowContent.setLayout(fl);

  // Create controls in memory
       JLabel label1 = new JLabel("Number 1:");
       JTextField field1 = new JTextField(10);
       JLabel label2 = new JLabel("Number 2:");
       JTextField field2 = new JTextField(10);
       JLabel label3 = new JLabel("Sum:");
       JTextField result = new JTextField(10);
       JButton go = new JButton("Add");

  // Add controls to the panel
       windowContent.add(label1);
       windowContent.add(field1);
       windowContent.add(label2);
       windowContent.add(field2);
       windowContent.add(label3);
       windowContent.add(result);
       windowContent.add(go);

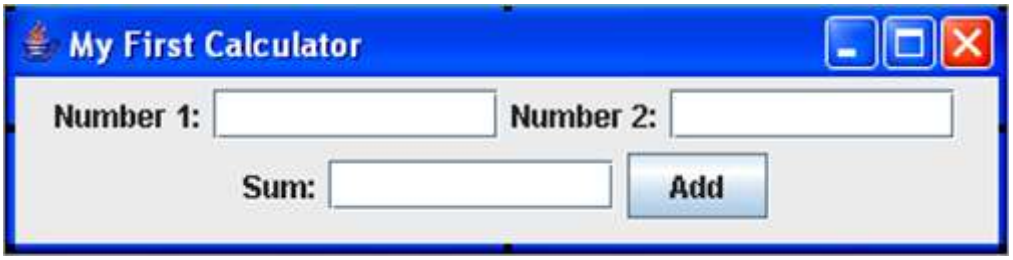
  // Create the frame and add the panel to it
  JFrame frame = new JFrame("My First Calculator");
```

```
    // Add the panel to the top-level container
    frame.setContentPane(windowContent);

    // set the size and make the window visible
    frame.setSize(400,100);
    frame.setVisible(true);
  }
}
}
```

Compile and run this program and it'll display the window shown in **Figure 8-2**. This may not be the best-looking calculator, but it demonstrates the use of `FlowLayout`. In the next section you'll make it look better with the help of more suitable layout managers.

**Figure 8-2**




Grab the corner of the window and make it wider. You'll see how `FlowLayout` starts reallocating controls, trying to fill the new area. If you make the window wide enough, all the components will fit in one row, as in **Figure 8-3**.

**Figure 8-3**




Even though you can enforce exact coordinates and sizes for each window component, Swing has layout managers that can maintain relative positions for all controls without assigning strict positions to

them. Layout managers will ensure that the content of a container looks nice regardless of the current window size.

### A Brief Introduction to Layout Managers

Swing offers the following layout managers:

- `FlowLayout`
- `GridLayout`
- `BoxLayout`
- `BorderLayout`
- `CardLayout`
- `GridBagLayout`

To use any layout manager, instantiate it first and then assign this instance to a container via `setLayout()` as you did with the class `SimpleCalculator` in **Listing 8-2**.

### FlowLayout

This layout arranges components in a window row by row. For example, labels, text fields, and buttons will be added to the first imaginary row until there is room left in this row. When the current row is filled, the rest of the components will go to the next row, and so on. If a user changes the size of the window, it may mess up the picture, as illustrated in **Figure 8-2**. Indeed, `FlowLayout` is not the best choice for our calculator. Let's try something different.

### GridLayout

The class `java.awt.GridLayout` allows you to arrange components as rows and columns in a grid. You'll be adding components to imaginary cells of this grid. If the container gets resized, grid cells may become larger or smaller, but the relative positions of the components inside the container will remain the same.

So far our calculator has seven components: three labels, three text fields, and a button. You may arrange them as a grid of four rows and two columns (one cell stays empty) by creating an instance of `GridLayout` like this:

```
GridLayout gr = new GridLayout(4,2);
```

You can also assign some horizontal and vertical spaces of, for example, five pixels, between the cells:

```
GridLayout gr = new GridLayout(4,2,5,5);
```

Replace `FlowLayout` with `GridLayout` in **Listing 8-2** and the calculator will look a little prettier. Create and compile a new class called `SimpleCalculatorGrid` (see **Listing 8-3**).




**Listing 8-3:** Calculator with GridLayout

```
import javax.swing.*;
import java.awt.GridLayout;
public class SimpleCalculatorGrid {
 public static void main(String[] args) {

       JPanel windowContent= new JPanel();

       // Set the layout manager for the panel
       GridLayout gl = new GridLayout(4,2);
       windowContent.setLayout(gl);
```

```
        JLabel label1 = new JLabel("Number 1:");
        JTextField field1 = new JTextField(10);
        JLabel label2 = new JLabel("Number 2:");
        JTextField field2 = new JTextField(10);
        JLabel label3 = new JLabel("Sum:");
        JTextField result = new JTextField(10);
        JButton go = new JButton("Add");

  // Add controls to the panel
        windowContent.add(label1);
        windowContent.add(field1);
        windowContent.add(label2);
        windowContent.add(field2);
        windowContent.add(label3);
        windowContent.add(result);
        windowContent.add(go);

   // Create the frame and add the panel to it
        JFrame frame = new JFrame("My First Calculator");

        frame.setContentPane(windowContent);

  // set the size and display the window
        frame.setSize(400,100);
        frame.setVisible(true);
   }
  }
```

Run the program `SimpleCalculatorGrid` and you'll see a much nicer calculator than before (see **Figure 8-4**).

**Figure 8-4**




Try to resize this window — controls will grow with the window, as shown in **Figure 8-5**, but their relative positions won't change. Note

that with `GridLayout` all cells of the grid have the same width and height.

Figure 8-5




### BorderLayout

The layout manager `java.awt.BorderLayout` divides a container into `South`, `West`, `North`, `East`, and `Center` areas. The `North` area stays on top of the window, `South` at the bottom, `West` on the left, and `East` on the right. For example, in the calculator shown in **Figure 8-6**, a text field that displays numbers is located in the `North` area, and the panel `p2` in the `West`.

You can use the following code to create a `BorderLayout` and place a text field there:

```
BorderLayout bl = new BorderLayout();
this.setLayoutManager(bl);

JTextField  txtDisplay = new JTextField(20);
this.add(BorderLayout.NORTH, txtDisplay);
```

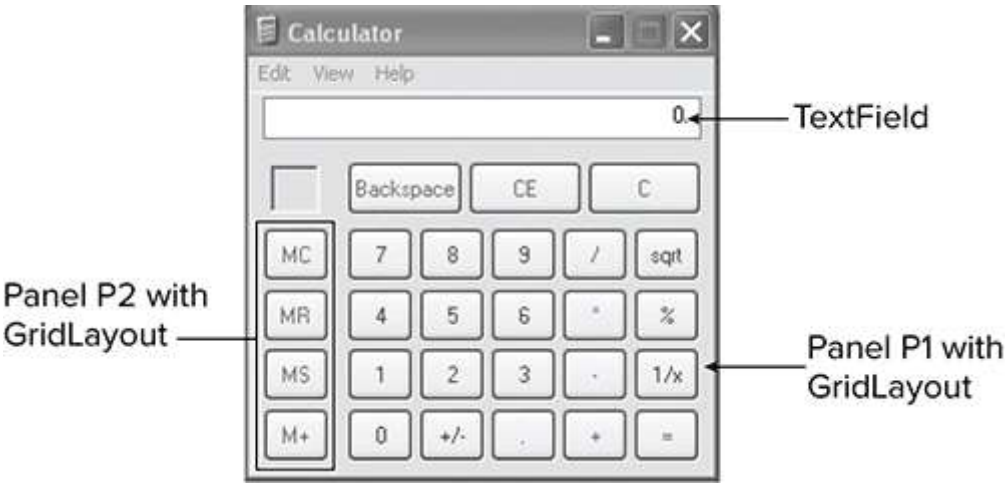
You are not required to have window controls in all five areas. If you need only `North`, `Center`, and `South` areas, the `Center` area will become wider because you are not going to use the `East` and `West` areas. I'll use a

`BorderLayout` a little later in this lesson, in the next version of our calculator: `Calculator.java`.

### Combining Layout Managers

Do you think that `GridLayout` will enable you to design a calculator that looks like the one that comes with Microsoft Windows, shown in **Figure 8-6**? Unfortunately it won't, because cells have different sizes there — the text field is wider than the buttons. You can, however, combine layout managers by using panels that have their own layout managers.

**Figure 8-6**

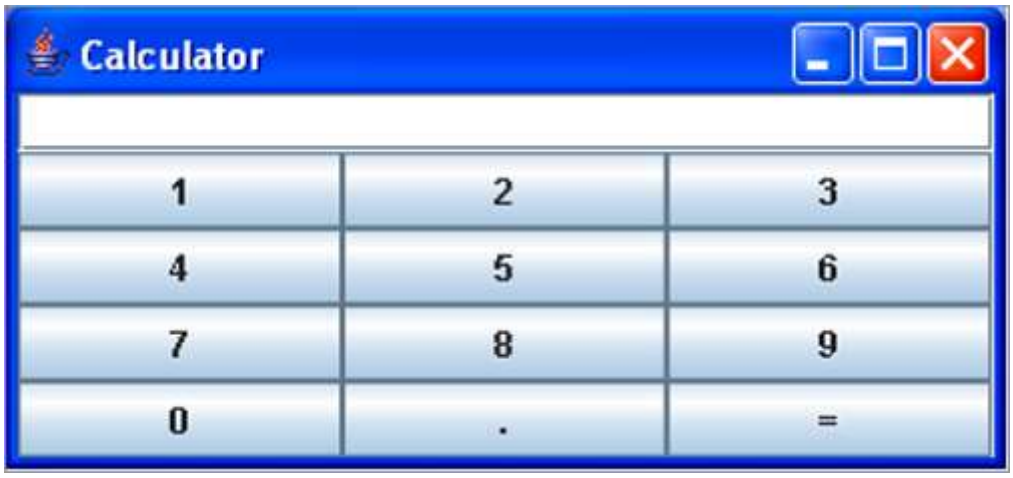



To combine layout managers in the new version of our calculator, do the following:

- Assign the border layout to the content panel of the frame.
- Add a `JTextField` to the `North` area of the frame to display the numbers.
- Create a panel, `p1`, with the `GridLayout`, add 20 buttons to it, and add it to the `Center` area of the content pane.
- Create a panel, `p2`, with `GridLayout`, add four buttons to the panel, and add `p2` to the `West` area of the content pane.

But first start by creating a somewhat simpler version of the calculator, which will look as shown in **Figure 8-7**.

Figure 8-7




Create a new class, `Calculator`, as per **Listing 8-4**, and run the program. Read the program comments — you should be able to understand how the code works by reading the comments, shouldn't you?

Available for download on Wrox.com

**Listing 8-4:** Calculator with combined layouts

```
import javax.swing.*;
import java.awt.GridLayout;
import java.awt.BorderLayout;

public class Calculator {

    // Declare all calculator's components.
       JPanel windowContent;
       JTextField displayField;
       JButton button0;
       JButton button1;
       JButton button2;
       JButton button3;
       JButton button4;
       JButton button5;
       JButton button6;
       JButton button7;
       JButton button8;
       JButton button9;
```

```
        JButton buttonPoint;
        JButton buttonEqual;
        JPanel p1;

       // Constructor  creates the components
       // and adds the to the frame using combination of
       // Borderlayout and Gridlayout

   Calculator(){

          windowContent= new JPanel();

         // Set the layout manager for this panel
          BorderLayout bl = new BorderLayout();
          windowContent.setLayout(bl);

        // Create the display field and place it in the
        // North area of the window

          displayField = new JTextField(30);
          windowContent.add("North",displayField);

      // Create buttons using constructor of the
      // class JButton that takes the label of the
      // button as a parameter

          button0=new JButton("0");
          button1=new JButton("1");
          button2=new JButton("2");
          button3=new JButton("3");
          button4=new JButton("4");
          button5=new JButton("5");
          button6=new JButton("6");
          button7=new JButton("7");
          button8=new JButton("8");
          button9=new JButton("9");
          buttonPoint = new JButton(".");
          buttonEqual=new JButton("=");

         // Create the panel with the GridLayout with 12 buttons –
         //10 numeric ones, period, and the equal sign

                 p1 = new JPanel();
                 GridLayout gl =new GridLayout(4,3);
                 p1.setLayout(gl);

         //  Add window controls to the panel p1
                 p1.add(button1);
                 p1.add(button2);
```

```
                p1.add(button3);
                p1.add(button4);
                p1.add(button5);
                p1.add(button6);
                p1.add(button7);
                p1.add(button8);
                p1.add(button9);
                p1.add(button0);
                p1.add(buttonPoint);
                p1.add(buttonEqual);

        // Add the panel p1 to the center of the window
                windowContent.add("Center",p1);

        //Create the frame and set its content pane
               JFrame frame = new JFrame("Calculator");
               frame.setContentPane(windowContent);

         // Set the size of the window big enough to accomodate all controls
               frame.pack();

                // Display the window
                 frame.setVisible(true);
      }

      public static void main(String[] args) {

          Calculator calc = new Calculator();
      }
 }
```

### BoxLayout

The class `javax.swing.BoxLayout` allows multiple window components to be laid out either horizontally (along the x-axis) or vertically (along the y-axis). Unlike with the `FlowLayout` manager, when the window with the `BoxLayout` is resized, its controls do not wrap. And unlike with `GridLayout`, with `BoxLayout`, window controls can have different sizes.

The next two lines of code assign `BoxLayout` with vertical alignment to `JPanel`. To make this code shorter, I have not declared a variable to store

a reference to the object `BoxLayout`, but rather create an instance of this object and immediately pass it to the method `setLayout()` as an argument.

```
JPanel p1= new JPanel();
setLayout(new BoxLayout(p1, BoxLayout.Y_AXIS));
```

If you just add several buttons to the panel `p1`, they all display one under another.

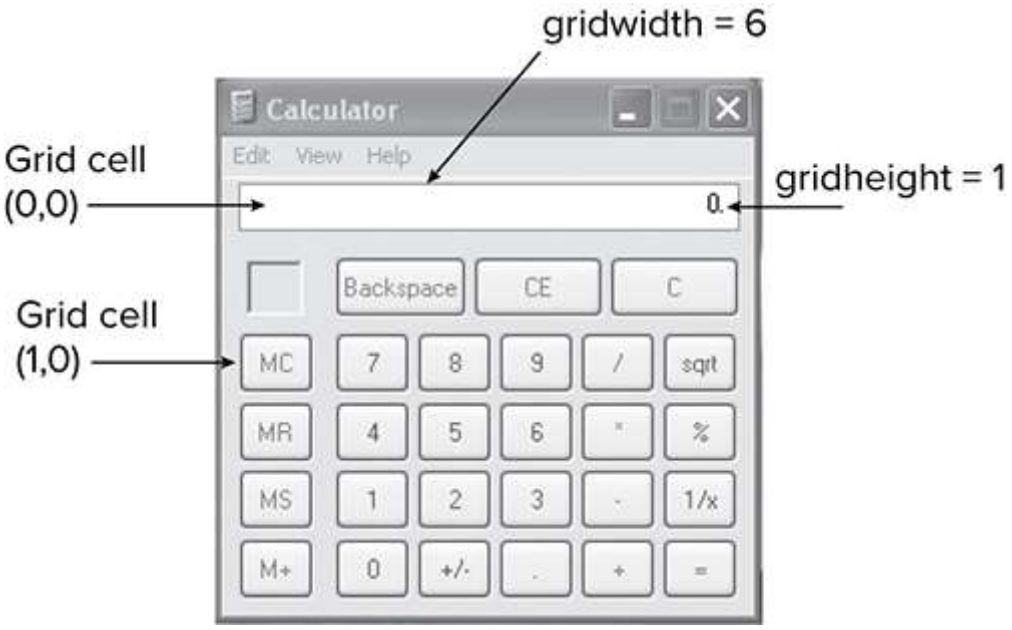
You can use combinations of various containers implementing horizontal or vertical `BoxLayout` to build a fairly sophisticated UI. Think of a front page of a game that has to have several items next to each other on the top of the window, some controls located vertically on the left sidebar, and the rest of the window's real estate allocated for the main battlefield. You can use `BorderLayout` having a panel with a horizontal `BoxLayout` on the North, and a panel with vertical `BoxLayout` on the West. The next section shows you a sophisticated yet more verbose `GridBagLayout`, but you should always try to see if the `BoxLayout` can do the job and use it, if possible.

### GridBagLayout

In this section you'll get familiar with yet another way of designing the calculator, by using the `java.awt.GridBagLayout` manager instead of combining panels with different layouts. `GridBagLayout` is an advanced grid that allows cells of different sizes. `GridBagLayout` works with another class called `GridBagConstraints`.

Constraints are just attributes of a cell, and you have to set them for each cell separately. All constraints for a cell have to be set before you place a component in the cell. For example, one of the constraint's attributes is called `gridwidth` (see **Figure 8-8**). It allows you to make a cell as wide as several other cells.

Figure 8-8




When working with the grid layout you should create an instance of the constraint object first, and then set the values to its properties. Then you can add a UI component to the cell in your container.

The code sample in Listing 8-5 is heavily sprinkled with comments to help you understand how to use `GridBagLayout`. While working on this lesson's assignment you'll be using this code.

Available for download on Wrox.com

Listing 8-5: Creating constraints for GridBagLayout

```
// Set the GridBagLayout for the window's content pane
 GridBagLayout gb = new GridBagLayout();
 this.setLayout(gb);

// Create an instance of the GridBagConstraints
// You'll have to repeat these lines for each component
// that you'd like to add to the grid cell
 GridBagConstraints constr = new GridBagConstraints();
```

```
//setting constraints for the Calculator's displayField:

// x coordinate in the grid
 constr.x=0;
// y coordinate in the grid
 constr.y=0;

// this cell has the same height as other cells
 constr.gridheight =1;

// this cell is as wide as 6 other ones
 constr.gridwidth= 6;

// fill all space in the cell
 constr.fill= constr.BOTH;
// proportion of horizontal space  taken by  this
// component
 constr.weightx = 1.0;

// proportion of  vertical space taken by  this component
 constr.weighty = 1.0;
// position of the component within the cell
 constr.anchor=constr.CENTER;

 displayField = new JTextField();
// set constrains for this field
 gb.setConstraints(displayField,constr);

// add the text field to the window
 windowContent.add(displayField);
```

### CardLayout

Think of a deck of cards lying on top of each other, only the top card visible. You can use the `java.awt.CardLayout` manager to create a component that shows one panel at a time, such as the tabbed folder in Figure 8-9.

Figure 8-9



When the user clicks a tab, the content of the window changes. In fact, all the panels needed for this screen are already preloaded and lie on top of each other. When the user clicks a tab, the program just brings this “card” to the top and makes the other “cards” invisible. The tabbed folder here was used for illustration; the Swing library includes a ready-to-go component for windows with tabs, called `JTabbedPane`.

### Containers with Absolute Layout

If you want a container’s content to remain the same regardless of the window’s size, set the x and y coordinates, width, and height (aka the bounds) of each component while adding them to the window. Your class has to explicitly state that it won’t use any layout manager by passing `null` to `setLayout()`:

```
windowContent.setLayout(null);
```

The next code snippet shows how you can set a button’s width to 40 pixels and its height to 20, and place it 100 pixels to the right of and 200 pixels down from the top left corner of the window:

```
JButton myButton = new Button("New Game");
myButton.setBounds(100,200,40,20);
```

### More about Swing Widgets

It’s not possible to describe all the Swing components in a short lesson — use the Swing online tutorial mentioned earlier in this lesson. Here’s a list of all the Swing widgets:

```
JButton

JLabel

JCheckBox

JRadioButton

JToggleButton

JScrollPane

JSpinner

JScrollBar

JSlider

JProgressBar

JComboBox

JList

JTabbedPane

JTable

JTextField

JTextArea

JPasswordField

JFormattedTextField

JEditorPane
```

`JToolTip`

`JTree`

`JViewPort`

`ImageIcon`

You can also create menus (`JMenu` and `JPopupMenu`), pop-up windows, and frames inside other frames (`JInternalFrame`), and use the standard-looking windows (`JFileChooser`, `JColorChooser`, and `JOptionPane`).

Java comes with an excellent demo application, SwingSet3, that shows all the available Swing components in action. You can find it in your JDK install directory. (On my computer it's in `C:\Program Files\Java\jdk1.6.0_19\demo\jfc\SwingSet3`.) Just open the file `readme.html` located in the same directory and it'll bring you to a web page containing instructions for launching the demo. The MAC OS users won't find these samples in their Java installation, but they are also available at **https://swingset3.dev.java.net**.

### Swing GUI Builders

Java developers use various tools to speed the process of designing UIs. See what's available for the IDE that you use. For example, there is a popular plug-in called Matisse that lets you design a Swing UI in what-you-see-is-what-you-get mode. You can find its version for Eclipse-based products, called MyEclipse, at the following URL: **www.myeclipseide.com/module-htmlpages-display-pid-1.html**.

Matisse was originally developed for the NetBeans IDE, and you can find it here: **http://netbeans.org/kb/trails/matisse.html**.

Finally, consider yet another Eclipse plug-in, called Jigloo GUI Builder: **http://marketplace.eclipse.org/content/jigloo-swtswing-gui-builder**. You can def-

initely find a tool that will substantially speed up your design of UIs with the Java Swing library.

### Try It

Your task for today is to create another version of the calculator in **Figure 8-8**, using only one layout: `GridBagLayout`.

### 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 Lesson8 folder in the download.

---

### Step-by-Step

This assignment comes down to creating appropriate constraints for each UI component shown in **Figure 8-8**. So there is just one long step in this assignment: Follow the example given in **Listing 8-5** for each UI component needed for the calculator.

---



Please select Lesson 8 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 9

# Event Handling in UI

Java Swing, like any other UI library, is an event-driven framework. When a user interacts with a GUI program (such as by clicking a button or pressing a key) a Java Swing program receives an **event** that can initiate an appropriate reaction.

If you write the code to react to a particular event, this code will be invoked. If you haven't written such code, the event will be fired anyway, but the program won't respond to it. In this lesson you'll learn how to handle events in Java GUI programs.

**Introduction to Event Listeners**

I'm sure you've tried to click the buttons of our calculator from Lesson 8, but they were not ready to respond to your actions yet. Swing widgets can process various events, or in the programmers' jargon can **listen to events**. To listen to events, a program has to register window components with Java classes called **listeners**.

You should have components listen only to the events they are interested in. For example, when a person clicks a button, it's not important where exactly the mouse pointer is as long as it is on the button's surface. That's why you do not need to register the button with `MouseMotionListener`. On the other hand, this listener comes in handy for all kinds of drawing programs.

To process button clicks Swing provides `ActionListener`. All listeners are declared as Java interfaces and their methods have to be implemented in an object that will be listening to events.

This is how Java documentation describes the `ActionListener` interface:

The listener interface for receiving action events. The class that is interested in processing an action event implements this interface, and the object created with that class is registered with a component, using the

component's addActionListener method. When the action event occurs, that object's actionPerformed method is invoked.

This interface is defined in the `java.awt.event` package, as presented in **Listing 9-1**.




**Listing 9-1:** ActionListener interface

```
public interface ActionListener extends EventListener
  void actionPerformed(ActionEvent e);

}
```

The `actionPerformed()` method is invoked by the JVM if the action happened. Let's use this listener in the calculator you created in Lesson 8.

### Teaching the Calculator to Calculate

The calculator's buttons should register themselves with a class that implements `ActionListener`, which means that its method `actionPerform()` contains the calculation logic. Even though you can implement `ActionListener` in the `Calculator` class itself, for better readability and code maintainability it's best to separate the code defining the UI from the code containing processing logic. Let's start writing a separate class, `CalculatorEngine`:

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

}
```

This class won't compile; Java will give an error message stating that the class must implement the method `actionPerformed(ActionEvent e)`. You know the rules for interfaces, right? The code in **Listing 9-2** will fix this error.




**Listing 9-2:** First implementation of ActionListener interface