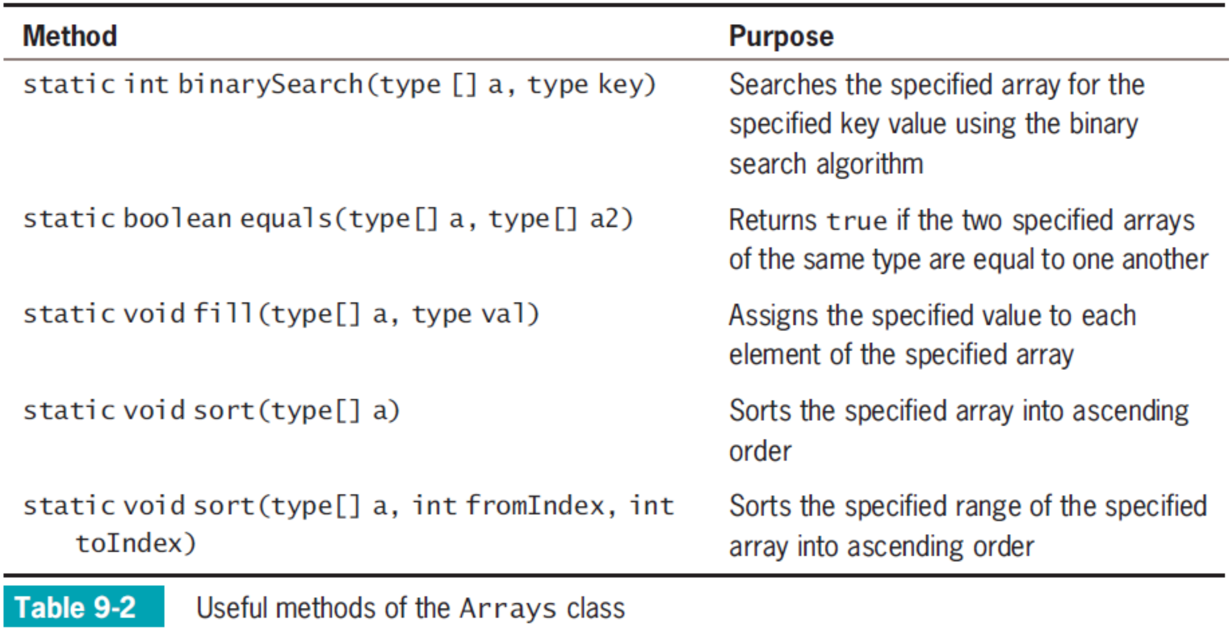
Chapter 3 - Implementing Classes

Using the `Arrays` Class

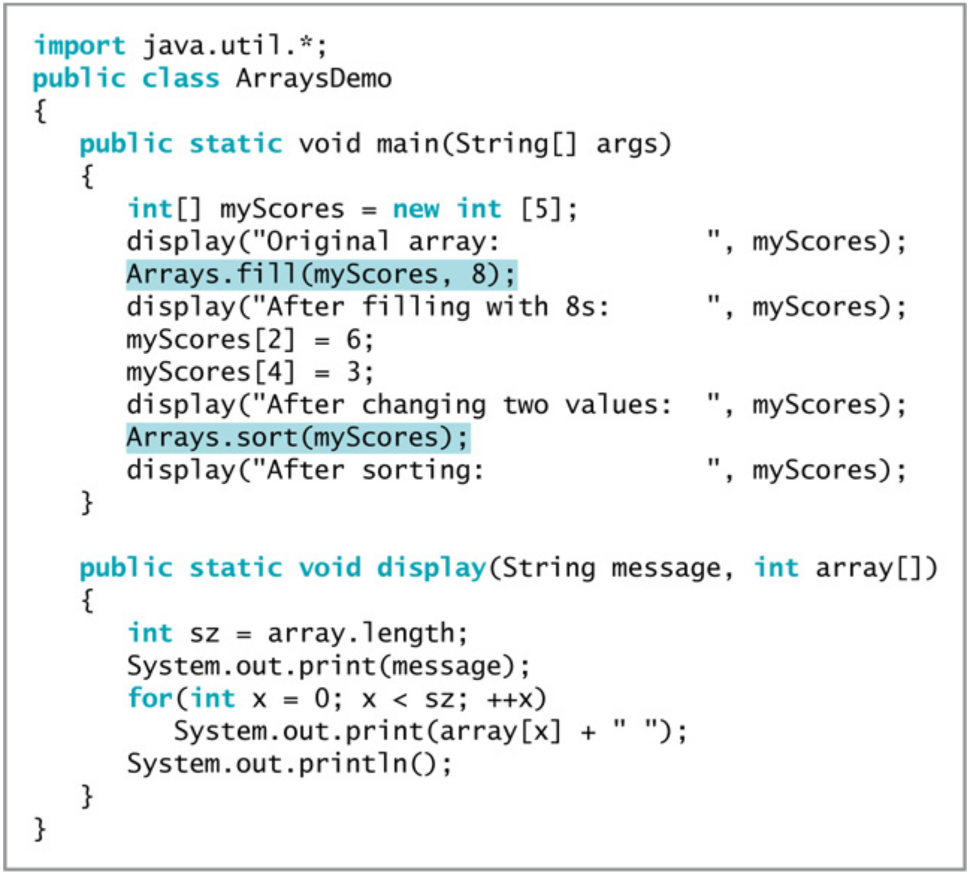
- **`Arrays` class**
  - Contains many useful methods for manipulating arrays
  - `static` methods
    - Use them with the class name without instantiating an `Arrays` object
  - `binarySearch()` method
    - A convenient way to search through sorted lists of values of various data types
    - The list must be in order

Using the `Arrays` Class (cont'd.)



Method	Purpose
`static int binarySearch(type [] a, type key)`	Searches the specified array for the specified key value using the binary search algorithm
`static boolean equals(type[] a, type[] a2)`	Returns `true` if the two specified arrays of the same type are equal to one another
`static void fill(type[] a, type val)`	Assigns the specified value to each element of the specified array
`static void sort(type[] a)`	Sorts the specified array into ascending order
`static void sort(type[] a, int fromIndex, int toIndex)`	Sorts the specified range of the specified array into ascending order

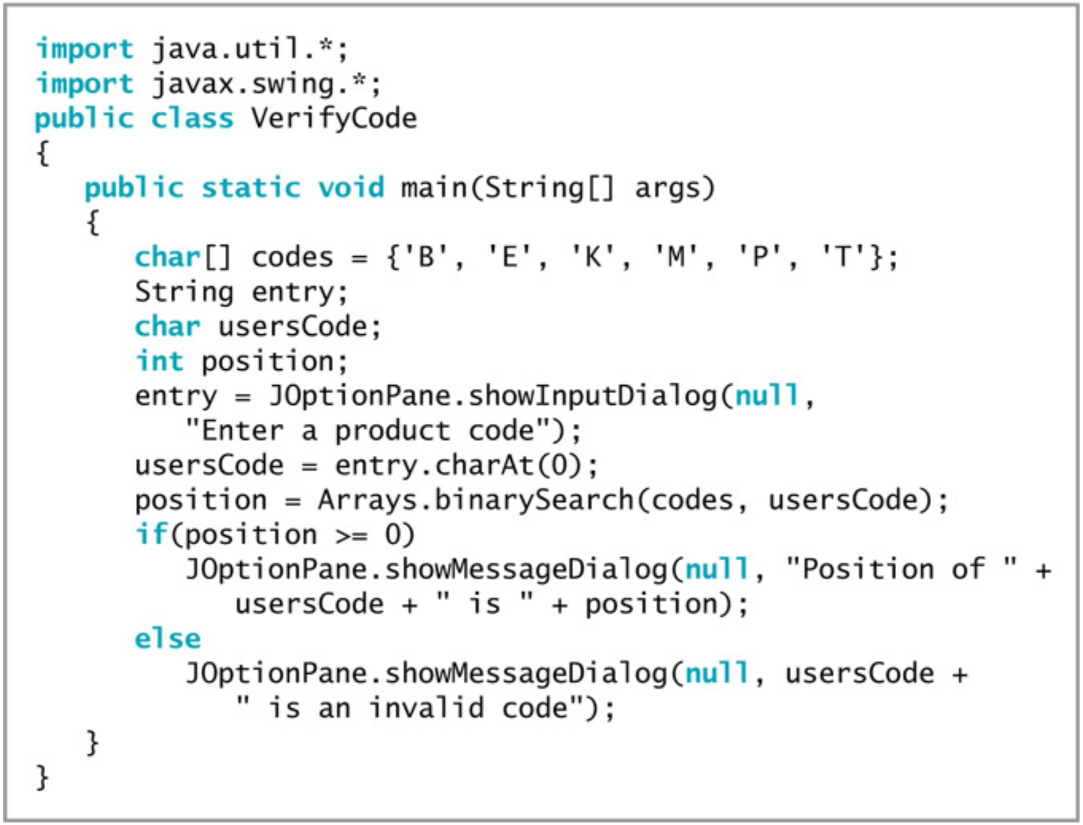
Table 9-2 Useful methods of the `Arrays` class



```java
import java.util.*;
public class ArraysDemo
{
   public static void main(String[] args)
   {
      int[] myScores = new int [5];
      display("Original array:             ", myScores);
      Arrays.fill(myScores, 8);
      display("After filling with 8s:      ", myScores);
      myScores[2] = 6;
      myScores[4] = 3;
      display("After changing two values:  ", myScores);
      Arrays.sort(myScores);
      display("After sorting:              ", myScores);
   }

   public static void display(String message, int array[])
   {
      int sz = array.length;
      System.out.print(message);
      for(int x = 0; x < sz; ++x)
         System.out.print(array[x] + " ");
      System.out.println();
   }
}
```

Figure 9-15 The `ArraysDemo` application



```
import java.util.*;
import javax.swing.*;
public class VerifyCode
{
   public static void main(String[] args)
   {
      char[] codes = {'B', 'E', 'K', 'M', 'P', 'T'};
      String entry;
      char usersCode;
      int position;
      entry = JOptionPane.showInputDialog(null,
         "Enter a product code");
      usersCode = entry.charAt(0);
      position = Arrays.binarySearch(codes, usersCode);
      if(position >= 0)
         JOptionPane.showMessageDialog(null, "Position of " +
            usersCode + " is " + position);
      else
         JOptionPane.showMessageDialog(null, usersCode +
            " is an invalid code");
   }
}
```

Figure 9-17 The `VerifyCode` application

Local Variables

- **Local variables** are declared in the body of a method:

```
public double giveChange()
{
   double change = payment - purchase;
   purchase = 0;
   payment = 0;
   return change;
}
```

- When a method exits, its local variables are removed.
- **Parameter variables** are declared in the header of a method:

```
public void enterPayment(double amount)
```

Local Variables

- Local and parameter variables belong to methods:
  - When a method runs, its local and parameter variables come to life
  - When the method exits, they are removed immediately
- Instance variables belong to objects, not methods:
  - When an object is constructed, its instance variables are created
  - The instance variables stay alive until no method uses the object any longer
- Instance variables are initialized to a default value:
  - Numbers are initialized to 0
  - Object references are set to a special value called `null`
    - A `null` object reference refers to no object at all
- You must initialize local variables:
  - The compiler complains if you do not

Self Check 3.21

What do local variables and parameter variables have in common? In which essential aspect do they differ?

Answer: Variables of both categories belong to methods – they come alive when the method is called, and they die when the method exits. They differ in their initialization. Parameter variables are initialized with the call values; local variables must be explicitly initialized.

The `this` Reference

- Two types of inputs are passed when a method is called:
  - The object on which you invoke the method
  - The method arguments
- In the call `momsSavings.deposit(500)` the method needs to know:
  - The account object (`momsSavings`)
  - The amount being deposited (`500`)
- The **implicit parameter** of a method is the object on which the method is invoked.
- All other parameter variables are called **explicit parameters**.

The `this` Reference

- Look at this method:

```
public void deposit(double amount)
{
   balance = balance + amount;
}
```

`amount` is the explicit parameter

The implicit parameter(`momSavings`) is not seen

`balance` means `momSavings.balance`

- When you refer to an instance variable inside a method, it means the instance variable of the implicit parameter.

The `this` Reference

- The `this` reference denotes the implicit parameter

```
balance = balance + amount;
```

- actually means

```
this.balance = this.balance + amount;
```

- When you refer to an instance variable in a method, the compiler automatically applies it to the `this` reference.

The this Reference

- Some programmers feel that inserting the `this` reference before every instance variable reference makes the code clearer:

```
public BankAccount(double initialBalance)
{
   this.balance = initialBalance;
}
```

The this Reference

- The `this` reference can be used to distinguish between instance variables and local or parameter variables:

```
public BankAccount(double balance)
{
   this.balance = balance;
}
```

- A local variable shadows an instance variable with the same name.

   You can access the instance variable name through the `this` reference.

- In Java, local and parameter variables are considered first when looking up variable names.
- Statement

```
this.balance = balance;
```

means: "Set the instance variable `balance` to the parameter variable `balance`".

The this Reference

- A method call without an implicit parameter is applied to the same object.
- Example:

```
public class BankAccount
{
   . . .
   public void monthlyFee()
   {
      withdraw(10); // Withdraw $10 from this account
   }
}
```

- The implicit parameter of the `withdraw` method is the (invisible) implicit parameter of the `monthlyFee` method
- You can use the `this` reference to make the method easier to read:

```
public class BankAccount
{
   . . .
   public void monthlyFee()
   {
      this.withdraw(10); // Withdraw $10 from this account
   }
}
```

Chapter 8 - Designing Classes

Packages

- **Package:** Set of related classes
- Important packages in the Java library:

Package	Purpose	Sample Class
`java.lang`	Language support	`Math`
`java.util`	Utilities	`Random`
`java.io`	Input and output	`PrintStream`
`java.awt`	Abstract Windowing Toolkit	`Color`
`java.applet`	Applets	`Applet`
`java.net`	Networking	`Socket`
`java.sql`	Database Access	`ResultSet`
`javax.swing`	Swing user interface	`JButton`
`omg.w3c.dom`	Document Object Model for XML documents	`Document`

Organizing Related Classes into Packages

© Don Wilkie/iStockphoto.

In Java, related classes are grouped into packages.

Organizing Related Classes into Packages

- To put classes in a package, you must place a line

  ```
  package packageName;
  ```

  as the first instruction in the source file containing the classes.

- Package name consists of one or more identifiers separated by periods.

- To put the `Financial` class into a package named `com.horstmann.bigjava`, the `Financial.java` file must start as follows:

  ```
  package com.horstmann.bigjava;

  public class Financial
  {
      . . .
  }
  ```

- A special package: default package

  Has no name

  No `package` statement

If you did not include any package statement at the top of your source file

- its classes are placed in the default package.

Importing Packages

- Can use a class without importing: refer to it by its full name (package name plus class name):

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

- Inconvenient
- `import` directive lets you refer to a class of a package by its class name, without the package prefix:

```
import java.util.Scanner;
```

- Now you can refer to the class as `Scanner` without the package prefix.
- Can import all classes in a package:

```
import java.util.*;
```

- Never need to import `java.lang`.
- You don't need to import other classes in the same package of the project.

Package Names

- Use packages to avoid name clashes:

  ```
  java.util.Timer
  ```

  vs.

  ```
  javax.swing.Timer
  ```

- Package names should be unique.
- To get a package name: turn the domain name around:

  ```
  com.horstmann.bigjava
  ```

- Or write your email address backwards:

  ```
  edu.sjsu.cs.walters
  ```

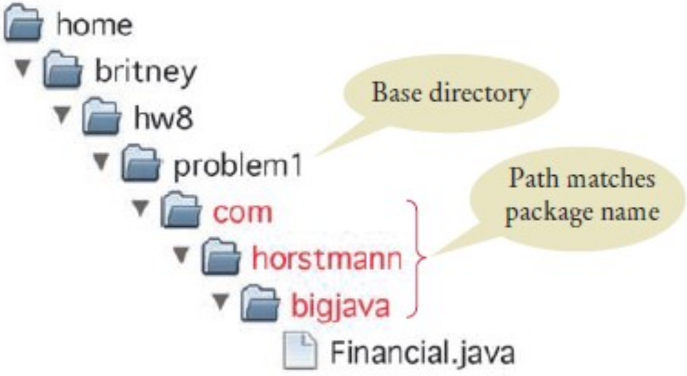


Figure 6 Base Directories and Subdirectories for Packages

Self Check 8.25

Which of the following are packages?

a. `java`

b. `java.lang`

c. `java.util`

d. `java.lang.Math`

Answer: (a) No; (b) Yes; (c) Yes; (d) No

Self Check 8.26

Is a Java program without `import` statements limited to using the default and `java.lang` packages?

Answer: No — you simply use fully qualified names for all other classes, such as `java.util.Random` and `java.awt.Rectangle`.

Self Check 8.27

Suppose your homework assignments are located in the directory `/home/me/cs101` (`c:\Users\me\cs101` on Windows). Your instructor tells you to place your homework into packages. In which directory do you place the class `hw1.problem1.TicTacToeTester`?

Answer: `/home/me/cs101/hw1/problem1` or, on Windows, `c:\Users\me\cs101\hw1\problem1`

Unit Test Frameworks

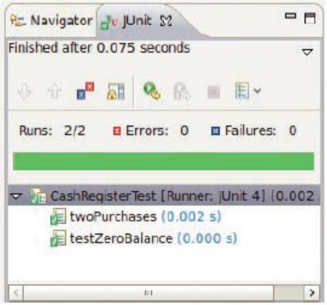
- Unit test frameworks simplify the task of writing classes that contain many test cases.
- JUnit: `http://junit.org`
    - Built into some IDEs like BlueJ and Eclipse
- Philosophy: whenever you implement a class, also make a companion test class. Run all tests whenever you change your code.

Unit Test Frameworks

- Customary that name of the test class ends in `Test`:

```
import org.junit.Test;
import org.junit.Assert;
public class CashRegisterTest
{
   @Test public void twoPurchases()
   {
      CashRegister register = new CashRegister();
      register.recordPurchase(0.75);
      register.recordPurchase(1.50);
      register.enterPayment(2, 0, 5, 0, 0);
      double expected = 0.25;
      Assert.assertEquals(expected, register.giveChange(), EPSILON);
   }
   // More test cases
   . . .
}
```

- If all test cases pass, the JUnit tool shows a green bar:



```
Finished after 0.075 seconds

Runs: 2/2    Errors: 0    Failures: 0

CashRegisterTest [Runner: JUnit 4] (0.002
   twoPurchases (0.002 s)
   testZeroBalance (0.000 s)
```

Figure 7 Unit Testing with JUnit

Self Check 8.29

What is the significance of the `EPSILON` parameter in the `assertEquals` method?

Answer: It is a tolerance threshold for comparing floating-point numbers. We want the equality test to pass if there is a small roundoff error.