Chapter 4

Classes, Objects, and Methods

OOP Concepts

Outline

String class
Writing classes that use strings
Writing classes that use association and aggregation
Using other library classes
Object construction
Object references
Data encapsulation and integrity
Running a class from the command line
4.1 Introduction

In this Chapter we continue with the basic concepts and terminology of object-oriented programming (OOP). One of the main themes is that existing library classes and classes we write ourselves can be used together as building blocks to create the classes we need to solve problems.

First we introduce the String class which represents sequences of characters and we write some classes that use the String class. We introduce other important library classes such as the Date, Calendar, SimpleDateFormat and NumberFormat classes.

The important OOP concepts of association and aggregation are explained using several examples, including a Point class and a Circle class.

Then we summarize and extend the important OOP concepts we have encountered so far. These include object construction, object references, comparison of primitive and object types, comparison of assignment statements for primitive and reference types, using references as arguments and method return values, data encapsulation and integrity, side-effects (both desirable and undesirable), comparison of instance and static variables and methods, and the call by value argument passing mechanism use by Java.

Other useful library classes such as the Calendar, Date, and SimpleDateFormat are also introduced.

We write several classes that show how to use the String class and that also illustrate the important ideas of association and aggregation. The BlueJ environment is very useful for exploring these concepts.

Finally, we introduce the main method which is necessary in order to run Java classes outside BlueJ from the command line.

4.2 String class

In Java the String class is part of a standard Java library (package) called java.lang and contains many methods for operating on string objects. Each String object is a sequence of 0 or more characters. This is one of the most important classes since strings are used in almost every program.

We will explore this class using BeanShell and BlueJ and use strings in our own classes just like we used the primitive int and double types in Chapter 3.

4.2.1 Constructing strings

A literal string is a sequence of characters delimited by double quote characters. For example, "Hello" is a literal string. The double quote characters are used to delimit the characters included in the string but are not part of the string. The length of a string is defined as the number of characters in the string, so this string has length 5.

The empty string has no characters and a length of zero. It is denoted by "", two consecutive double quote characters with nothing between them.

Each character in a string is a Unicode character of type char stored internally as a 16-bit integer code. The char data type was briefly discussed in Chapter 2. To use strings we normally do not need to understand how characters are represented internally as integers.
Each character in a string can be directly referenced by an index. The index begins at zero so the first character has index 0, the next has index 1, and so on. The string "Hello" can be represented by the picture

```
0 1 2 3 4
H e l l o
```

Here each character is shown in a box and its index is shown below the box. The index can be used to specify individual characters in the string. In the picture the first l has index 2 (or position 2).

A **substring** of a given string is a string constructed from a subsequence of characters from the string beginning at one index position and ending at another. In the picture the substring "ell" begins at index 1 and ends at index 3.

To construct the literal string object "Hello Fred" and give it the name `greeting` we use the declaration statement

```java
String greeting = "Hello Fred";
```

Strings are special, we do not need to use `new` to create a `String` object as we did in Chapter 3 (see Table 3.1 for example).

It is very important to understand that `greeting` is defined on the left side of this declaration as the name of a `String` object, not the `String` object itself. We say that `greeting` is an **object reference variable**. The right side creates the `String` object and returns a reference to it which is then assigned to `greeting`. Loosely speaking we often say that `greeting` is an object when we really mean that it is an object reference variable.

A picture of this process is shown in Figure 4.1. The box named `greeting` holds the reference,

```
Figure 4.1: A string object in memory and a reference to it called greeting
```

shown as an arrow, and the rounded box represents the memory reserved for the object and the characters in the string. This is very different from the way the primitive types such as `int` and `double` work (compare with Figure 2.2)

### 4.2.2 String expressions and concatenation

The most common string operation is **concatenation**. It takes two strings and joins them together to make a new string. It is also called **append** since it appends one string to the end of another. In
Java the + operator is used to denote string concatenation and it should not be confused with its use in the addition of numbers. If \( s_1, s_2, \ldots, s_n \) are either strings or expressions that can be converted to strings then

\[ s_1 + s_2 + \ldots + s_n \]

is called a string expression and it is always evaluated from left to right unless there are parentheses. We can use BeanShell to illustrate string construction and concatenation.

**Example 4.1** (Constructing literal strings) The statements

```bash
bsh % String name = "William " + "James " + "Duncan";
bsh % print(name);
William James Duncan
```

show how to concatenate three literal strings to make a new string.

**Example 4.2** (String variable expressions) Generalizing the preceding example, define the string variables \( \text{first}, \text{middle} \) and \( \text{last} \) for three names and concatenate them into a full name using the statements

```bash
bsh % String first = "William";
bsh % String middle = "James";
bsh % String last = "Duncan";
bsh % String fullName = first + " " + middle + " " + last;
bsh % print(fullName);
William James Duncan
```

The last declaration concatenates five strings, two of which are strings consisting of a single space. The three variables are replaced by their string values.

**String expressions containing numbers**

String expressions that contain both strings and numbers are very convenient. They are called mixed string expressions and are evaluated like pure string expressions except that any numbers or other expressions are converted to strings before the results are concatenated together.

**Example 4.3** (Mixed string expressions) In the BeanShell statements

```bash
bsh % String area = "Area: " + Math.sqrt(2.0);
bsh % print(area);
Area: 1.4142135623730951
```

the square root function returns a double value which is converted to a string and then concatenated with the literal string.

There are some pitfalls to watch out for, as the following example shows.
4.2 String class

**Example 4.4** (Parentheses in string expressions) Suppose that \( x \) and \( y \) are integer variables that have the values 3 and 4, respectively. The evaluation of the string expression

"The sum of \( " + x + " \) and \( " + y + " \) is \( " + x + y \)

gives the string expression

"The sum of 3 and 4 is 34"

which is probably not what you want. To obtain the desired result

"The sum of 3 and 4 is 7"

it is necessary to use parentheses as in

"The sum of \( " + x + " \) and \( " + y + " \) is \( " + (x + y) \)

so that the final + is interpreted as an addition instead of a string concatenation.

4.2.3 String methods

Many methods are available in the `String` class. For example, we can find the length of a string, the character at a specified position, or a substring. As we did in Chapter 2 for the functions in the `Math` class, we can document the string methods using their prototypes.

**The length of a string**

The prototype for the length method is

```
public int length()
```

indicating that this is an instance method that returns the number of characters in the string. Since string indices begin at zero, a valid index \( i \) for a string \( s \) should be in the range

\[
0 \leq i \leq s.length() - 1
\]

**Example 4.5** (Length of a string) The BeanShell statements

```
bsh % String name = "Harry";
bsh % int len = name.length();
bsh % print(len);
5
```

use the instance method call expression `name.length()` to assign the length of `name` to the integer variable `len`. 

---
Converting a number to a string

**EXAMPLE 4.6** (Converting numbers to strings) If `age` is an integer variable and `area` is a double precision variable then the statements

```java
String s1 = "" + age;
String s2 = "" + area;
```

convert the numbers to strings by using the empty string to force `+` to be interpreted as concatenation. This trick forces numeric values to be converted to strings.

**Extracting a single character from a string**

Sometimes we need to extract specific characters from a string. There is an instance method called `charAt` to do this: It has the prototype

```java
public char charAt(int index)
```

The return value is the character at the index (beginning at 0) as specified by the argument.

**EXAMPLE 4.7** (Extracting a character from a string) The statements

```bash
bsh % String s = "Hello";
bsh % char c = s.charAt(1);
bsh % print(c);
e
bsh % print(s.charAt(5));
// Error: // Uncaught Exception: ...
... java.lang.StringIndexOutOfBoundsException: String index out of range: 5
```

show how to assign the character at position 1 in the string "Hello" to a `char` value and display it. The last statement shows that if you specify an index outside the valid range (0 to 4 here), a `StringIndexOutOfBoundsException` error message is displayed.

**Constructing a substring**

The `substring` method constructs a new string that is a substring of a given string. There are two versions with prototypes

```java
public String substring(int firstIndex)
public String substring(int firstIndex, int lastIndexPlusOne)
```

These instance methods are used to send messages to a string object and return one of its substrings. The first version has one formal argument and returns the substring beginning at index `firstIndex` and continuing to the end of the string. The second version has two formal arguments. It returns the substring starting at index `firstIndex` and ending at index `lastIndexPlusOne - 1` rather than the index of the last character of the substring as you might expect.

Since the `String` class is immutable the original string is unchanged. The substring operation creates a new string.
4.2 String class

**Example 4.8** (Substrings) Let \( d \) be an integer variable with a value in the range 1000 to 999999. The BeanShell statements

```bash
bsh % int d = 531452;
bsh % String sd = "" + d;
bsh % int len = sd.length();
bsh % sd = "$" + sd.substring(0,len-3) + "," + sd.substring(len-3);
bsh % print(sd);
$531,452
```
convert it to a string in the range \( \$1,000 \) to \( \$999,999 \). The second substring expression extracts the digits that go after the comma and the first extracts the digits that precede the comma.

**Trimming a string**

The `trim` method can be used to remove leading and trailing spaces from a string. It is an instance method with the prototype

```
public String trim()
```

A new string that has no leading or trailing spaces is created by this method and a reference to it is returned.

**Example 4.9** (Trimming a string) The BeanShell statements

```bash
bsh % show();
<true>
bsh % String s, t;
bsh % s = " Hello ";
< Hello >
bsh % t = s.trim();
<Hello>
```
create \( t \), a trimmed version of \( s \).

**Upper case and lower case conversions**

There are methods to convert strings from lower case to upper case and vice versa. The prototypes are

```
public String toLowerCase()
public String toUpperCase()
```

The first method can be used to construct a lowercase version of a string with all uppercase letters replaced by lowercase ones, and the second method does the opposite.

**Example 4.10** (Case conversion) The BeanShell statements
Classes, Objects, and Methods

bsh % String test = "Hello";
bsh % String upper = test.toUpperCase();
bsh % print(upper);
HELLO

creates an upper case version of the string test. Similarly, the statements

bsh % String test = "transfer";
bsh % char first = test.toUpperCase().charAt(0);
bsh % print(first);
T

return the upper case version of the first character of the string test.

Searching for substrings

Sometimes it is useful to know if one string is a substring of another one. There are four instance methods to do this. Their prototype are

    public int indexOf(int ch)
    public int indexOf(int ch, int startIndex)
    public int indexOf(String sub)
    public int indexOf(String sub, int startIndex)

The first two methods search for a character ch in a string and the last two methods search for a string sub. The one argument versions begin the search at the start of the string and the two argument versions begin at startIndex.

In any case the value returned is −1 if the character or substring is not found. Otherwise the index of the character or the first character of the substring is returned.

Also notice that these four methods all have the same name. This is an example of method overloading. The compiler can determine which version to use by looking at the signature of the method: the types and number of the arguments.

[EXAMPLE 4.11] (Searching using indexOf) The BeanShell statements

bsh % String indices = "0123456789012345678901234567890";
bsh % String target = "This is the target string";
bsh % print(target.indexOf('u'));
-1
bsh % print(target.indexOf('g'));
15
bsh % print(target.indexOf("the"));
8
bsh % print(target.indexOf("target",12));
12
bsh % print(target.indexOf("target",13));
-1
bsh %
show how to search for characters and substrings.

## 4.2 String class

### 4.2.4 Displaying numbers and strings

In BeanShell we have been using either `show` or `print` to display numbers and strings. In BlueJ we have dialog boxes that appear and show the result (return value) of invoking a method that returns a value.

These display techniques are not really part of Java. However Java can produce what is called **console output** or **terminal output**, if you have a console or terminal window, using an object called `System.out` which is automatically provided by the Java interpreter. It has two methods called `print` and `println` for displaying data.

Some of the method prototypes are

```java
public void println(int n)
public void println(double d)
public void println(String s)
public void println()
```

Here `println` stands for “print line”. There are many different `println` methods having different argument types. We have shown four common ones here. This is another example of method overloading. In each case the value of the actual argument is converted to a string and displayed on a line in the terminal window and the next output will begin on a new line. The `println` method with no argument simply moves to the next line.

For each `println` method there is a corresponding `print` method. The difference is that after a `print` method is executed the next output will begin on the same line.

### Using System.out in BeanShell

In BeanShell if you select “Capture System in/out/err” from the File menu and then you can use `System.out`.

**EXAMPLE 4.12** (Using `System.out` in BeanShell) Try the statements

```java
double area = 3.14159;
System.out.println("Area: "+ area);
System.out.print("Area: ");
System.out.println(area);
```

in the BeanShell editor and see the results in the workspace. The last two statements produce the same output as the first `println` statement.

### Special characters in strings

The `println` statement

```java
System.out.println("Hello");
```
has the same effect as the `print` statement

```java
System.out.print("Hello\n");
```

The backslash used in a string is called an **escape character** and its use affects the meaning of the next character. In this example it means to interpret \n as the “newline” character instead of its literal meaning. Thus, the newline character can be expressed as '\n'. It is most useful in print statements since each time it is used it causes a line break to the beginning of the next line.

There are several of these special characters. For example, the carriage return character is denoted by '\r' and the tab character is denoted by '\t'.

If you need to include a backslash literally in a string then it is necessary to use two backslashes like this: `\`. Or, you may want to use the double quote character literally in a string. This can be done using `\"` in the string to specify that the double quote character is not the string delimiter.

### Example 4.13 (Using backslash as an escape character)

The statement

```java
System.out.println("\\\Hello\\\n");
```

displays the string

"Hello"

since two double quotes and two backslash characters are included in the string.

### Using `System.out` in BlueJ

Using `System.out` or BeanShell’s `show` and `print` functions is the only way to test classes outside BlueJ but we don’t normally need it in BlueJ. However BlueJ has a terminal window and if you use `System.out` it will automatically appear showing the output. We will find this useful later. For now, here is a small example.

### Example 4.14 (Using `System.out` in BlueJ)

Add the method

```java
public void display()
{
    System.out.println("Radius = " + radius);
    System.out.println("Area = " + area);
    System.out.println("Circumference = " + circumference);
}
```

to the CircleCalculator class (page 63). Now create an object for a radius of 3, choose its display method and you will see the terminal window shown in Figure 4.2.
4.2 String class

The `toString` method

The `toString` method is a special method in Java with the prototype

```java
public String toString()
```

The purpose of this method is to return a string representation of an object of the class. An interesting property of this method is that if you use the name of an object in a string expression the object will be replaced by the result of calling its `toString` method. If the class does not contain a `toString` method a default one will be used.

**EXAMPLE 4.15** (Using `toString`) The BeanShell statements

```bash
bsh % addClassPath("c:/book-projects/chapter3");
bsh % CircleCalculator circle = new CircleCalculator(3.0);
bsh % String rep = "toString gives " + circle;
bsh % print(rep);
toString gives CircleCalculator@4d1d41
bsh % rep = "toString gives " + circle.toString();
bsh % print(rep);
toString gives CircleCalculator@4d1d41
bsh % print(circle);
CircleCalculator@4d1d41
```

show what the default string representation looks like for a `CircleCalculator` object. It is not very useful. Notice that using the name `circle` is equivalent to using `circle.toString()` so the `toString` method essentially defines how an object can be converted to a string. The last statement shows that if the name of an object is used as an argument to `print` or `System.out.print` in Java then the object will be converted to a string value using the `toString` method.

**Defining our own `toString` method**

The default `toString` method is not very useful but we can redefine it in any of our classes and our version will be called.

**EXAMPLE 4.16** (Adding `toString` to `CircleCalculator`) Add the method

```java
public String toString()
```
public String toString()
{
    return "CircleCalculator[radius=" + radius + ", area=" +
    area + ", circumference=" + circumference + "]";
}

to the CircleCalculator class (page 63). Now create an object for a radius of 3, choose its
toString method and you will see the result in a “Method Result” box.

**Example 4.17** (Trying it with BeanShell) Since we modified the class make sure you start
a new version of BeanShell before trying the statements

```
bsh % addClassPath("c:/book-projects/chapter3");
bsh % CircleCalculator circle = new CircleCalculator(3.0);
bsh % print(circle);
CircleCalculator[radius=3.0, area=28.274333882308138,
circumference=18.84955592153876]
```

which show our string representation for a radius of 3.

The toString method is sometimes useful for finding logical errors in your classes. If you
want to see the state of an object at a given place in the execution of a class you can insert a
statement of the form

```
System.out.println(obj);
```

where obj is the name of the object.

### 4.2.5 Formatting numbers and strings (Java 5)

In Java 5 (Java 1.5) it is possible to format items (numbers and strings and other objects) according
to the specifications provided in a format string. This gives precise control over how many
columns are used for each item (field width), whether an item is left or right justified within its
field width, and how many digits are displayed after the decimal point in the case of floating point
numbers.

The String class contains a static method with prototype

```
public static String format(String f, Object... args)
```

Here f is the format string and Object... args represents the values to be formatted.

There is also a printf method that can be used with a format string that has the prototype

```
public void printf(String f, Object... args)
```

These methods are available in the latest version of BlueJ but not in BeanShell.

**Example 4.18** (Format codes) Here are some useful format codes (each begins with %).
4.2 String class

%5d      format an integer right justified in field of width 5
%-5d     format an integer left justified in field of width 5
%-20s    format a string left justified in a field of width 20
%15.5f   format a floating point number right justified in a field of width 15
         using fixed format rounded to 5 digits after the decimal point
%.5f     format a floating point number in a field that just fits using fixed format
         rounded to 5 digits after the decimal point
%20.8e   format a floating point number right justified in a field of width 20
         using exponential (scientific) format rounded to 8 digits after the
decimal point

There are many other types of codes. A complete list can be found in the Java documentation for
the Formatter class.

Example 4.19 (Formatted strings) The statements

    int i = 3;
    double pi = Math.PI;
    String end = "End";
    String f = String.format("answer: %5d%15.5f%10s", i, pi, end);
    System.out.println(f);

produce the output

    answer: 3 3.14159 End

consisting of the literal string "answer: " followed by the integer 3 right justified in a field of
width 5 followed by the value of π right justified in a field of width 15 and rounded to 5 digits after
the decimal point, and followed by the value of the string end right justified in a field of width 10.

The printf method can also be used to specify the format string and print it. The statements

    int i = 3;
    double pi = Math.PI;
    String end = "End";
    System.out.printf("answer: %5d%15.5f%10s
", i, pi, end);

produce the same output if we add the newline character \n to the end of the format string.

Example 4.20 (Using System.out.printf in BlueJ) Repeat Example 4.14 using the
following method.

    public void display()
    {
        System.out.printf("Radius = %.5f\n", radius);
        System.out.printf("Area = %.5f\n", area);
        System.out.printf("Circumference = %.5f\n", circumference);
    }
to display the values rounded to 5 digits.

4.3 Example classes that use the String class

We now write some simple classes that use the String class. We also illustrate a three step “design, implement, test” process for writing classes.

1. Begin with an English description of the class.

2. Design the class by deciding what methods it should have. This is called writing the class specification or the public interface. If the class is designed to be used primarily by other classes (the String class is an example) then check the convenience of your design by writing some typical statements that use the class.

3. Write the complete class by providing the implementation. This involves choosing any instance data fields, and providing bodies for all methods and even providing private methods, if necessary, to aid in the implementation.

4. Test the class by itself using BlueJ or BeanShell, or both. Even if the class is designed to be used by other classes it should be tested by itself before being used in a larger system of classes. This is called unit testing.

4.3.1 BankAccount class (first version)

A description of this class is

“A BankAccount object should represent a bank account using an account number, an owner name, and a current balance. There should be a constructor for creating a bank account given these values. There should be methods to withdraw or deposit a given amount and the usual “get methods” for returning the account number, owner name, and balance.”

This is a mutable class since the withdraw and deposit methods change the balance in the account. We use this class many times throughout the book to illustrate important concepts.

Designing the class

The English description directly gives the following specification or public interface.

```java
public class BankAccount
{
    // put instance data field declarations here
    public BankAccount(int accountNumber, String ownerName, double initialBalance) {...}
    public void deposit(double amount) {...}
    public void withdraw(double amount) {...}
```
4.3 Example classes that use the String class

```java
BankAccount myAccount = new BankAccount(123, "Peter Pascoe", 125.50);

public BankAccount(int accountNumber, String ownerName, double initialBalance)
{
   ...
}
```

![Figure 4.3: Matching actual and formal constructor arguments](image)

```java
public int getNumber() {...}
public String getOwner() {...}
public double getBalance() {...}
```

We have used the notation {...} to indicate that the method bodies are part of the implementation step, not the public interface step. Also we have not shown the javadoc comments but they should also be included in this step. We have not indicated the instance data fields yet.

For this simple class it is easy to check our design. For example, to create an account, withdraw $100, and show the current balance we could use the statements

```java
BankAccount account = new BankAccount(123, "Peter Pascoe", 125.50);
account.withdraw(100);
System.out.println("The current balance is " + account.getBalance());
```

The correspondence between the formal arguments in the constructor prototype

```java
public BankAccount(int accountNumber, String ownerName, double initialBalance)
```

and the actual arguments in the constructor call expression

```java
new BankAccount(123, "Peter Pascoe", 125.50)
```

is shown in Figure 4.3.

**Implementing the class**

We need to declare three private instance data fields for the account number, account owner name, and current balance:

```java
private int number;
private String name;
private double balance;
```
There is one set of these variables for each bank account object and the purpose of the constructor is to initialize these fields. Therefore, we can write the constructor as

```java
public BankAccount(int accountNumber, String ownerName, double initialBalance) {
    number = accountNumber;
    name = ownerName;
    balance = initialBalance;
}
```

The deposit method needs to add the amount specified by the formal argument to the value of the instance data field for the balance and the withdraw method needs to subtract this amount. Therefore, we can write the deposit method as

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

and the withdraw method

```java
public void withdraw(double amount) {
    balance = balance - amount;
}
```

The “get methods” are easy to write since they just return the value of one of the instance data fields. For example,

```java
public String getName() {
    return name;
}
```

We can also add a toString method. Here is the completed class declaration complete with javadoc comments.

```java
package chapter4.bank_account; // remove this line if you’re not using packages

/**
 * First version of the BankAccount class. No error checking is performed.
 * There is a better version in library.BankAccount.
 * Each object from this class encapsulates the account number, owner name, and current balance of a bank account.
 */
```
4.3 Example classes that use the String class

```java
/*
public class BankAccount
{
    private int number;
    private String name;
    private double balance;

    /**
     * Construct a bank account with given account number, owner name
     * and initial balance.
     * @param accountNumber the account number.
     * @param ownerName the owner name.
     * @param initialBalance the initial balance.
     */
    public BankAccount(int accountNumber, String ownerName,
                        double initialBalance)
    {
        number = accountNumber;
        name = ownerName;
        balance = initialBalance;
    }

    /**
     * Deposit the given amount of money in the account.
     * @param amount the amount to deposit (no error checking).
     */
    public void deposit(double amount)
    {
        balance = balance + amount;
    }

    /**
     * Withdraw the given amount of money from the account.
     * @param amount the amount to withdraw (no error checking).
     */
    public void withdraw(double amount)
    {
        balance = balance - amount;
    }

    /**
     * Return the account number.
     * @return the account number.
     */
    public int getNumber()
    {
        return number;
    }

    /**
     * Return the owner name.
     * @return the owner name.
     */
```
*/
public String getName()
{
    return name;
}

/**
 * Return the account balance.
 * @return the account balance.
 */
public double getBalance()
{
    return balance;
}

/**
 * Return a string representation of a bank account.
 * @return a string representation of a bank account.
 */
public String toString()
{
    return "BankAccount[number=" + number + ", name=" + name + ", balance=" + balance + "]";
}

Testing the class

This is only a preliminary version of this class. Later when we have introduced conditional statements we can make a more robust version of this class that checks, for example, if the amount specified in the withdraw method would not result in an overdrawn account.

This class is easy to test in BlueJ. For example, construct an object called circle as shown in Figure 4.4(a) and choose some of its methods from the method menu shown in Figure 4.4(b).

You can also test this class using either the BeanShell workspace or the BeanShell editor.

**EXAMPLE 4.21 (Testing BankAccount with BeanShell)** The following statements

```bash
bsh % addClassPath("c:/book-projects/chapter4/bank_account");
bsh % BankAccount account = new BankAccount(123, "Peter Pascoe", 125.50);
bsh % account.withdraw(100);
bsh % print(account.getBalance());
25.5
bsh % account.deposit(100);
bsh % print(account.getBalance());
125.5
bsh % print(account);
BankAccount[number=123, name=Peter Pascoe, balance=125.5]
```

show how to create an account, withdraw and deposit $100, and use the toString method to display the account.
4.3 Example classes that use the String class

Testing is important even in a simple class like this. While writing this book the author used the `toString` method

```java
public String toString()
{
    return "BankAccount[number=\" + name + \", name=\"
    + name + \", balance=\" + balance + \"]";
}
```

Can you find the logical error? The error was noticed when BeanShell produced the result

```bash
bsh % print(account);
BankAccount[number=Peter Pascoe, name=Peter Pascoe, balance=125.5]
```

### 4.3.2 `InitialsMaker` class

A description of this class is

> “An InitialsMaker object uses the first and last name of a person to produce the initials. For example, if the name is Henry James then the initials are HJ.”

#### Designing the class

We need a constructor that has two string arguments for the first and last names, a method to return the initials as a string and we will also include a `toString` method.

This gives the public class interface

```java
public class InitialsMaker
{
```
Classes, Objects, and Methods

// instance data fields go here
public InitialsMaker(String firstName, String lastName) {...}
public String getInitials() {...}
public String toString() {...}
}

We choose to make this an immutable class so no “set methods” are provided.

Implementing the class

The class is easily implemented. We choose one instance data field of type String for the initials:

    public String initials

We have made a decision here not to include the first and last names as data fields, so we do not include “get” methods for them. An alternate design would be to provide these data fields and their associated “get methods”.

The constructor can use the substring method to extract the first letter of each name. Then these letters can be concatenated together to get a two character string. Finally, this string can be converted to upper case:

    public InitialsMaker(String firstName, String lastName)
    {
        initials = firstName.substring(0,1) + lastName.substring(0,1);
        initials = initials.toUpperCase();
    }

The complete class is given by

```java
package chapter4.strings; // remove this line if you’re not using packages

/**
 * An object of this class takes a person’s first and last names
 * and extracts the first letter of each name
 * to makes a two character initial string converted to upper case.
 */
public class InitialsMaker
{
    private String initials; // string containing two initials

    /**
     * Construct an object for the given names
     * @param firstName the first name
     * @param lastName the last name
     */
    public InitialsMaker(String firstName, String lastName)
    {
```
4.3 Example classes that use the String class

```java
initials = firstName.substring(0,1) + lastName.substring(0,1);
initials = initials.toUpperCase();
```

```java
/**
 * Return the initial string
 * @return return the initial string
 */
public String getInitials()
{
    return initials;
}

public String toString()
{
    return "InitialsMaker[initials=" + initials + "]";
}
```

Testing the class

The class is easily tested in both BlueJ and BeanShell. There are two kinds of tests: (a) are the first letters of each name being extracted properly, and (b) are the initials being converted to upper case properly. For example, names like "Fred Duncan" can test (a) but not (b), where it is necessary to try names such as “henry james”, “Henry james”, and "henry James”.

**EXAMPLE 4.22** (Testing InitialsMaker with BeanShell) The following statements

```bash
bsh % addClassPath("c:/book-projects/chapter4/strings");
bsh % InitialsMaker maker = new InitialsMaker("harry", "james");
bsh % print(maker.getInitials());
HJ
bsh % print(maker);
InitialsMaker[initials=HJ]
```

show how to create an object and perform one of the tests on it.

4.3.3 PasswordGenerator class

A description of this class is

“A PasswordGenerator object generates random 7 character passwords. The first four characters should be lower case letters and the last three characters should be digits 0 to 9.”

Designing the class

Since no input is required we only need the constructor with no arguments. Also we need a method called next which will return the next random password as a string each time it is called. This gives the simple class interface
public class PasswordGenerator
{
    // instance data fields go here
    public PasswordGenerator() {...}
    public String next() {...}
}

Implementing the class

Before implementing this class we need to find out how to generate random numbers. Either we have to do it ourselves or we find out if Java can do it. Fortunately, Java can do it so we will follow the “do not reinvent the wheel policy”.

In Example 2.27 (page 29) we showed how to use the Math.random method in the Math class. This method can generate random double numbers in the range \(0 \leq r < 1\) which were converted to integers in the range \(1 \leq i \leq 10\). We could use this approach but there is also a class called Random whose objects can generate random integers directly. It has two constructors with prototypes

```java
public Random()
public Random(long seed)
```

The first constructor is used to generate a sequence of random numbers that depends on the current time in milliseconds. In other words the same sequence will not be repeated. The second constructor generates sequences that use a “seed”. Each value of the seed gives a repeatable sequence. Of course if you are using a random number generator in a game you will not want to use this version since every time you run the game the same sequence will be generated.

This suggests that we modify our design to use two constructors. The no-arg constructor can use the current time to generate a seed and the other one can use a specified seed. This gives the modified class design

```java
public class PasswordGenerator
{
    // instance data fields go here
    public PasswordGenerator() {...}
    public PasswordGenerator(long seed) {...}
    public String next() {...}
}
```

This is our first example of a class that has more than one constructor. This is permissible and quite common as long as the constructors can be distinguished by their argument types (signature).

The Random class has several methods but the one we are interested in has the prototype

```java
public int nextInt(int n)
```

which generates a random integer \(i\) in the range \(0 \leq i \leq n - 1\). We now have enough information to write the following partial implementation of the class

```java
public class PasswordGenerator
{
```
4.3 Example classes that use the String class

private Random random;

// any other instance data fields go here

public PasswordGenerator()
{
    random = new Random();
}
public PasswordGenerator(long seed)
{
    random = new Random(seed);
}
public String next() {...}

Here we have an object reference random being used as an instance data field. It refers to a Random object in the same way that name in the BankAccount class (page 106) refers to a String object. Thus, each constructor needs to create an object and assign its reference to random.

To implement the next method we need to first generate random characters in the range ‘a’ to ‘z’. This can be done by generating random integers in the range 0 to 25 using the method call expression random.nextInt(26) and using the result as an index into the string

String LETTERS = "abcdefghijklmnopqrstuvwxyz";

to generate a random letter. Starting with an empty string we have

    int index;
    String password = "";
    index = random.nextInt(26);
    password = password + LETTERS.substring(index, index+1);

Repeating the last two statements three more times gives us a string of four random letters.

To generate random digit characters we can simply use random.nextInt(10) and convert the result to a digit character by concatenation onto the letter string.

The LETTERS string is an example of a constant string and PasswordGenerator objects do not need their own copies of this string. One copy for all objects suffices and this can be indicated using the static modifier. Static data fields are not instance data fields:

private static final String LETTERS = "abcdefghijklmnopqrstuvwxyz";

This gives the following complete class declaration:

```
package chapter4.strings; // remove this line if you’re not using packages
import java.util.Random;
//**

Class PasswordGenerator

```
* An object of this class knows how to generate
* random 7 character password of the form LLLLLDD where L
* is a lower case letter and D is a digit.
*/
public class PasswordGenerator
{
    private static final String LETTERS = "abcdefghijklmnopqrstuvwxyz";
    private Random random;

    /**
     * Construct an default generator whose sequence is based on the
     * current time in milliseconds.
     */
    public PasswordGenerator()
    {
        random = new Random();
    }

    /**
     * Construct a generator that is repeatable. If the same seed
     * is used again then the same sequence is generated.
     * @param seed a seed to start the random number generator.
     */
    public PasswordGenerator(long seed)
    {
        random = new Random(seed);
    }

    /**
     * Return a generated password. Another password is generated each
     * time this method is called.
     * @return another password
     */
    public String next()
    {
        int index; // index into a string (0,1,2,...)

        String password = "";

        index = random.nextInt(26); // 0 to 25 inclusive
        password = password + LETTERS.substring(index, index + 1);
        index = random.nextInt(26);
        password = password + LETTERS.substring(index, index + 1);
        index = random.nextInt(26);
        password = password + LETTERS.substring(index, index + 1);
        index = random.nextInt(26);
        password = password + LETTERS.substring(index, index + 1);
        index = random.nextInt(10);
        password = password + index;
        index = random.nextInt(10);
        password = password + index;
        return password;
    }
}
4.3 Example classes that use the *String* class

```java
index = random.nextInt(10);
password = password + index;

return password;
```

**The import statement**  The very first line of this class is new. The `Random` class is not a standard class like `Math`, `System`, and `String`. These classes are in a Java package (collection of related classes) called `java.lang` that is automatically imported into any class that needs them. Thus, for example, the fully qualified name of the `String` class is `java.lang.String`.

For Java classes in other packages such as `java.util` it is necessary to explicitly import them. For `Random` this is done using

```java
import java.util.Random;
```

before the class declaration.

It is not essential to use the import statement. Its purpose is simply to allow you to use the short names instead of the fully qualified ones. Thus, if `Random` is not imported then we must use its full name `java.util.Random` everywhere. For example, as a private data field we would need to write

```java
private java.util.Random random;
```

and in the constructors we would need to write

```java
random = java.util.Random();
random = java.util.Random(seed);
```

**Testing the class**

This class is easily tested either with BeanShell or with BlueJ. The following example shows how to test the class in BeanShell.

```bash
EXAMPLE 4.23 (Testing PasswordGenerator with BeanShell )  The following statements

```

```

bsh % addClassPath("c:/book-projects/chapter4/strings");
bsh % PasswordGenerator gen = new PasswordGenerator();
bsh % print(gen.next());
avfi637
bsh % print(gen.next());
igde665
bsh % gen = new PasswordGenerator(); // make a new one
bsh % print(gen.next());
zuwe456
bsh % gen = new PasswordGenerator(123);
bsh % print(gen.next());
```
show how to create objects and test them. Four PasswordGenerator objects are created here and references to them are assigned to the reference variable gen.

The first two are based on the current time so they produce different values the first time their next method is called but the last two use the same seed so the first time their next method is called the same password is produced.

With BlueJ try constructing objects of each type (current time or seed) and try the next method on the object menu several times. Another approach is to add the following test method to the PasswordGenerator class:

```java
public void test()
{
    System.out.println(next());
}
```

Each time we select this method from the object menu a random password is appended to the BlueJ terminal window.

**The “this” object**

There is an important idea in the above test method. Notice that next() is used without applying it on an object, whereas in BeanShell Example we used gen.next() since we already had the variable gen. The BeanShell statements were not part of the PasswordGenerator class. They were simply statements that used the class.

However, the next() method belongs to the PasswordGenerator class itself and we do not have an explicit object name like gen to refer to. In a situation like this we use what is called the this object. Using next() is equivalent to using this.next().

We have seen this before in the QuadraticRootFinder class (page 67). There it was used to refer to an instance data field when an argument had the same name. Now we are using it to refer to “this” object. The compiler will also accept

```java
System.out.println(this.next());
```

where this is explicit but it is not necessary.

Whenever you see an instance method call expression in a class that is not prefixed with an object name, this is the implied object. In the doCalculations method in QuadraticRootFinder this was implied and we could have used

```java
this.doCalculations()
```
4.4 Association and aggregation

Our Java programs normally consist of interacting objects from several classes. These classes are either built-in classes such as `String` that come with the Java SDK (software development kit), classes obtained from someone else, such as your course instructor or friends, or classes we write ourselves such as `CircleCalculator` or `PasswordGenerator`. We will also see that complex classes are commonly designed in terms of simpler ones. The terms association and aggregation are often used to describe how classes can relate to each other.

4.4.1 Association

Some classes are not related to any other classes. For example, the classes in Chapter 3 do not require any other class in order be compiled and work properly.

On the other hand, the `PasswordGenerator` class (page 113) depends on both the `String` and `Random` classes. If either of these classes were not available then the class could not be compiled. We say that the `PasswordGenerator` class is associated with, or uses, these classes. This relationship is not symmetric so we do not say that the `String` class is associated with the `PasswordGenerator` class. Classes, such as the `String` class, are designed to be used by other classes.

Thus, class A is associated with class B if A uses B. This can occur in several ways:

1. An object of B is used as a local variable in a constructor or method in A.
2. An object of B is used as a method or constructor argument in A.
3. An object of B is used as a return value of a method in A.
4. An object of B is used as an instance data field in A.

When we use the word object here we really mean an object reference. We will see many examples of these four kinds of relationships.

4.4.2 Aggregation

Case (4) above is an important special case of association called aggregation. This is how we make complex objects out of simpler ones and is often used in a “bottom up” approach to object-oriented design. First we design, implement, and test the simplest classes and then we use them as instance data fields of more complex classes, and so on.

We have already seen some simple examples of aggregation in the classes in this Chapter that use `String` objects as instance data fields and the `PasswordGenerator` class that uses a `Random` object.

4.4.3 TriangleCalculatorTester class

As an example of association in which one class uses a local variable of another class consider the following small class that could be used to test the `TriangleCalculator` class.
Classes, Objects, and Methods

### Class TriangleCalculatorTester

```java
package chapter4.tester; // remove this line if you're not using packages
import chapter3.TriangleCalculator; // remove this line if you're not using packages
/**
 * A short class to show how to test the TriangleCalculator
 * class from Chapter3 using System.out.println
 */
public class TriangleCalculatorTester
{
    public TriangleCalculatorTester()
    {
    }

    /**
     * Test the TriangleCalculator class
     * @param a side length
     * @param b another side length
     * @param g contained angle in degrees
     */
    public void doTest(double a, double b, double g)
    {
        TriangleCalculator tri = new TriangleCalculator(a,b,g);
        System.out.println("Sides: "+tri.getA()+", "+tri.getB()
        + ", " + tri.getC());
        System.out.println("Angles: "+tri.getAlpha()+", "+tri.getBeta()
        + ", " + tri.getGamma());
        System.out.println("Angle sum is " + tri.checkAngleSum());
    }
}
```

In this class the constructor has nothing to do so its body is empty. The `doTest` method does all the work by creating a `TriangleCalculator` object as a local variable and displaying the results in the terminal window.

This class is in BlueJ project `book-projects/chapter4/tester`, as shown in Figure 4.5(a). We have also included with the `TriangleCalculator` class from Chapter 3(page 64) which was in the `book-projects/chapter3` project. The dashed line from the `TriangleCalculatorTester` class to the `TriangleCalculator` class indicates the association or "uses" relation. The results of constructing an object with \( a = 1, b = 1 \) and \( c = 90 \) degrees is shown in the terminal window in Figure 4.5(b).

#### 4.4.4 Point class

As an another example of association and the idea of building objects from simpler objects we first consider a `Point` class for geometrical points \((x,y)\). Then we will use it to write a `Circle` class.
4.4 Association and aggregation

Designing the class

The Point class has instance data fields of type double for the x and y coordinates of a point, constructors, ”get methods” for the coordinates, and we include a toString method. We also assume that this class is immutable. Therefore, the public interface is

```java
public class Point
{
    double x, y;
    public Point() {...}
    public Point(double x, double y) {...}
    public double getX() {...}
    public double getY() {...}
    public String toString() {...}
}
```

For the constructor with no arguments we choose to construct the point at the origin with coordinates (0,0). Many other methods could be included (see end of Chapter exercises).

Implementing the class

Here is the complete implementation of the class.

```java
package chapter4.geometry; // remove this line if you’re not using packages
/**
 * A class representing immutable geometrical points (x,y)
 * in the plane.
 */
```
public class Point {
    private double x;
    private double y;

    /**
     * Construct a point from its coordinates.
     * @param x the x coordinate of the point
     * @param y the y coordinate of the point
     */
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    /**
     * Construct the default point (0,0).
     */
    public Point() {
        x = 0.0;
        y = 0.0;
    }

    /**
     * Return the x coordinate of this point.
     * @return the x coordinate of this point
     */
    public double getX() {
        return x;
    }

    /**
     * Return the y coordinate of this point.
     * @return the y coordinate of this point
     */
    public double getY() {
        return y;
    }

    /**
     * Return a string representation of a Point.
     * @return a string representation of a Point
     */
    public String toString() {
        return "Point[" + x + ", " + y + "]";
    }
}
Testing the class

This class is easily tested in BlueJ. With BeanShell we have to be careful as the following example shows.

EXAMPLE 4.24 (Testing Point with BeanShell) The following statements

```bash
bsh % addClassPath("c:/book-projects/chapter4/geometry");
bsh % import Point; // necessary or we get java.awt.Point
bsh % Point origin = new Point();
bsh % Point p = new Point(1,2);
bsh % print(origin);
Point[0.0, 0.0]
bsh % print(p);
Point[1.0, 2.0]
bsh % print(p.getX());
1.0
bsh % print(p.getY());
2.0
```

show how to create some Point objects and test them. It is necessary here to use the import statement.

4.4.5 Circle class

The Circle class describes a circle in terms of its radius and the x and y coordinates of its center.

Designing the class

Our first attempt at an interface for this class might be

```java
public class Circle
{
    private double x, y, radius;
    public Circle() {...}
    public Circle(double x, double y, double r) {...}
    public double getX() {...}
    public double getY() {...}
    public double getRadius() {...}
    public String toString() {...}
}
```

1 There is already a class in package java.awt called Point and BlueJ will use it by default unless we tell it to use our version. Therefore we need the import statement.
Here we have two constructors. The no-arg constructor is for the unit circle, with radius 1 and center (0,0). The next one uses the x and y coordinates of the center and the radius to define the circle.

In this design we are not associating the `Circle` class with the `Point` class so there would be no association between the two classes. However, since we already have the `Point` class it would be better to use aggregation and use the instance data fields and interface given by

```java
public class Circle
{
    private Point center;
    private double radius;
    public Circle() {...} // no-arg constructor
    public Circle(double x, double y, doubler r) {...} // constructor with x, y, and r
    public Circle(Point c, double r) {...} // constructor with Point object and r
    public Point getCenter() {...} // get center as a Point object
    public double getRadius() {...} // get radius
    public String toString() {...} // string representation
}
```

Now we choose 3 constructors. The first two have the same prototypes as in the previous design but now the third one allows a `Point` object to be used as a reference argument. Now we have a `getCenter` method that returns a reference to the center of the circle as a `Point` object. We could have also included the `getX` and `getY` method but they can be obtained using the corresponding methods of the `Point` class applied to the `Point` object returned by `getCenter`.

**Implementing the class**

Here is the completed `Circle` class.

```java
package chapter4.geometry; // remove this line if you’re not using packages
/**
 * A class representing immutable geometrical circles.
 * Each circle is described by its center (a Point object)
 * and its radius (a double number).
 */
public class Circle
{
    private Point center;
    private double radius;

    /**
     * Construct circle with given center point and radius.
     * @param p the center of the circle
     * @param r the radius of the circle
     */
```
public Circle(Point p, double r)
{
    center = p;
    radius = r;
}

/**
 * Construct circle with given center coordinates and radius.
 * @param x the x coordinate of the circle center
 * @param y the y coordinate of the circle center
 * @param r the radius of the circle
 */
public Circle(double x, double y, double r)
{
    center = new Point(x,y);
    radius = r;
}

/**
 * Construct a default circle: a unit circle with center (0,0)
 * and radius 1.
 */
public Circle()
{
    center = new Point();
    radius = 1;
}

/**
 * Return radius of circle.
 * @return radius of circle
 */
public double getRadius()
{
    return radius;
}

/**
 * Return center of circle.
 * @return center of circle
 */
public Point getCenter()
{
    return center;
}

/**
 * Return a string representation of a Circle.
 * @return a string representation of a Circle
 */
public String toString()
```java
return "Circle[" + center + ", " + radius + "]";
}
}

Recall that when an object name is used in a string it is replaced by the result of calling its `toString` method. We have used this to include the `toString` method of the `Point` class in the `toString` method of the `Circle` class.

### Testing the class with BeanShell

The following example shows how to test the `circle` class in BeanShell.

**[EXAMPLE 4.25] (Testing Circle with BeanShell)** Try the following statements in BeanShell:

```bash
bsh % addClassPath("c:/book-projects/chapter4/geometry");
bsh % import Point; // necessary or we get java.awt.Point
bsh % Point center = new Point(3,4);
bsh % Circle c1 = new Circle();
bsh % Circle c2 = new Circle(center, 5);
bsh % Circle c3 = new Circle(3,4,5);
bsh % print(c1);
Circle[Point[0.0, 0.0], 1.0]
bsh % print(c2);
Circle[Point[3.0, 4.0], 5.0]
bsh % print(c3);
Circle[Point[3.0, 4.0], 5.0]
bsh % double radius = c2.getRadius();
bsh % double x = c2.getCenter().getX();
bsh % double y = c2.getCenter().getY();
bsh % print(radius);
5.0
bsh % print(x);
3.0
bsh % print(y);
4.0
```

Here we construct a default circle `c1` and a circles `c2` and `c3` with center at `(3, 4)` and radius 5 in two ways using the second and third constructors. Then we use the `toString` method to display the results and show how the “get” methods are used. This also shows that we did not need to provide “get” methods in the `Circle` class for the `x` and `y` coordinates of the center since, for example, the `x` coordinate can be obtained using the method call expression `c2.getCenter().getX()` which uses two methods in one expression. First we get the center as `c2.getCenter()`. Since this is an object of type `Point` we can then invoke its `getX()` method all in one statement, and similarly for the `y` coordinate. When aggregation is involved it is common to see several method invocations strung together like this. In this case the separate statements...
4.4 Association and aggregation

Figure 4.6: Aggregation with the Point and Circle classes.

```java
Point c = c2.getCenter();
double x = c.getX();
double y = c.getY();
```

could also be used.

**Testing the class with BlueJ**

Because the Circle class uses aggregation it is interesting to test it in BlueJ. To do this construct four objects as follows so that you obtain the project shown in Figure 4.6.

1. Right click on the Point rectangle and select menu item `new Point(x,y)` to construct a point (3,4) with name center.

2. Right click on the Circle rectangle and select menu item `new Circle()` to construct a default circle with name c1.

3. Right click on the Circle rectangle again and select menu item `new Circle(p,r)` to construct a circle with name c2. Choose center as the center point. You can do it in the ‘Create Object’ dialog box by typing its name in the box, or just click in the text field for the center point and then click on the center object and its name should appear in the text field. Finally, choose 5 for the radius.

4. As above but choose menu item `new Center(x,y,r)` and enter 3, 4, and 5 for the values x, y, and r.

You should now have the four objects on the workbench as shown in Figure 4.6. The fact that the Circle class “uses” the Point class is indicated by the dashed arrow.

Now you can right click on any of the objects and choose a method or the “inspect” menu choice to test the class. To see aggregation and object references in action try the following
Classes, Objects, and Methods

Figure 4.7: (a) Choosing “Inspect” from the c2 menu, (b) choosing “Inspect” for the resulting object reference.

(a) (b)

Figure 4.8: (a) Choosing getCenter from c2 menu, (b) result of using ”get” from the “Method Result” box.

(a) (b)

1. Right click on object c2 and choose the “Inspect” menu choice to get the “Inspector” box shown in Figure 4.7(a). Instead of showing the Point object fields we only see <object reference>. This is BlueJ’s way of telling us that center is a reference to the object, not the object itself.

2. Now click on the this reference and click the “Inspect” button and you will actually see the x and y fields for the center object.

The Circle class uses aggregation by having a Point object as an instance data field but it also uses association in another way because the getCenter method returns a reference to the Point object that is the center of the circle. BlueJ lets you put such objects on the work bench as follows.

1. From the c2 object menu select the getCenter method to get the “Method Result” box shown in Figure 4.8(a).

2. Click on the object reference and click the “Get” button to get the box shown in Figure 4.8(b) asking for the name of the object.

3. Choose a name and the object will appear on the object bench.

We can also write classes, similar to TriangleCalculatorTester (page 118), to test the Point and Circle classes. Here is a simple example:
package chapter4.geometry; // remove this line if you’re not using packages
/**
 * A short class to show how to test the Circle and Point classes.
 */
public class CircleTester
{
  public CircleTester()
  {
  }

  /**
   * Test the Point and Circle classes.
   */
  public void doTest()
  {
    Point center = new Point(3,4);
    Circle c1 = new Circle();
    Circle c2 = new Circle(center, 5);
    Circle c3 = new Circle(3, 4, 5);
    System.out.println("c1 = " + c1);
    System.out.println("c2 = " + c2);
    System.out.println("c3 = " + c3);

    double radius = c2.getRadius();
    double x = c2.getCenter().getX();
    double y = c2.getCenter().getY();
    System.out.println("Radius = " + radius);
    System.out.println("Center x = " + x);
    System.out.println("Center y = " + y);
  }
}

4.5 Other library classes

We have used the String and Random library classes. There are many others that are useful and we illustrate a few of them here.

4.5.1 Dates and times

There are three classes that are useful for working with dates and formatting them.

Date class

The Date class in package java.util has two constructors and two useful methods, among others, with prototypes.
The first constructor creates a `Date` object for “right now”. In this class each date is represented by a `long` integer that is the number of milliseconds since January, 1, 1970, 00:00:00 GMT. For example, while I am typing this the date is 1055681816162. The `toString` method gives the more meaningful result “Sun Jun 15 08:56:56 EDT 2003”.

**Example 4.26** (The `Date` class using BeanShell)

```java
bsh % import java.util.Date;
bsh % Date now = new Date();
bsh % print(now);
Sun Jun 15 08:56:56 EDT 2003
bsh % long t = now.getTime();
bsh % print(t);
1055681816162
bsh % Date first = new Date(0L);
bsh % print(first);
Wed Dec 31 19:00:00 EST 1969
bsh % first.setTime(0L + 1000L * 60L * 60L * 24L);
bsh % print(first);
Thu Jan 01 19:00:00 EST 1970
bsh %
```

The `import` statement is necessary here. Also recall that a `long` integer literal needs an `L` suffix to distinguish it from an `int` literal. The first date corresponds to time 0, indicating that we are 5 hours behind GMT. Then 1 day is added to this time by adding to `0L` the number of milliseconds in a day.

**SimpleDateFormat class**

The `SimpleDateFormat` class in package `java.text` can be used to format dates in many different ways. Two of its constructors and a format method have prototypes

```java
public SimpleDateFormat()
public SimpleDateFormat(String pattern)
public String format(Date d)
```

This class is associated with the `Date` class. The first constructor uses a default format specific to the user locale and the second constructor has an argument to specify the format. The `format` method formats a given `Date` object as a `String` using the specified format.

**Example 4.27** (Using `SimpleDateFormat`)

Continuing the previous example try the following statements in BeanShell

```java
bsh %
```
import java.text.SimpleDateFormat;
SimpleDateFormat f1 = new SimpleDateFormat();
String n1 = f1.format(now);
print(n1);
6/15/03 8:56 AM

SimpleDateFormat f2 = new SimpleDateFormat("dd/MM/yyyy");
String n2 = f2.format(now);
print(n2);
15/06/2003

SimpleDateFormat f3 = new SimpleDateFormat("HH:mm:ss z");
String n3 = f3.format(now);
print(n3);
08:56:56 EDT

Again the import statement is necessary here. This example shows three of the many different date formats that are possible.  

Calendar class

The Date class does not deal with the many properties of dates such as the year, month, day of the month, day of the year, etc. The Calendar class in package java.util, has this functionality.

There are several different calendars in use around the world and Java can use any of them. In particular we are interested in the class called GregorianCalendar. Constructors for the Calendar class would be quite complex and would have to deal with the different conventions such as Sunday being the first day of the week in North American countries but Monday being the first day of the week in France.

Therefore we do not have constructors in the Calendar class. Instead a static getInstance method is provided to return the appropriate calendar object for your locale. Here are a few of the many methods in the Calendar class.

```java
public static Calendar getInstance()
public Date getTime()
public int get(int field)
public void set(int field, int value)
public void set(int year, int month, int day)
```

Recall that using static as a modifier on a method such as getInstance means that the method is not invoked on a particular object. Instead a static method is invoked using the class name. In our case we would use Calendar.getInstance(). This is like using Math.sqrt(2) in the Math class. Therefore, to construct a Calendar object for the current locale and the current time we would use a statement such as

```java
Calendar now = Calendar.getInstance();
```

See the Java documentation for more examples.
The `getTime` method can be used to convert the current calendar to a `long` integer time.

Since there are so many instance data fields to “get” the designers opted for one “get” method and an integer value to indicate what field to get rather than a separate method for each field. For example, the field number for the year is the constant `Calendar.YEAR`. There are similar constants for the many other fields. Static constants in another class are always accessed using the syntax

```
ClassName.constantName
```

There are two forms of the `set` method. The first can set an individual field to a value. For example to set the day of the month field to 1 (first day) we could use

```
calendar.set(Calendar.DAY_OF_MONTH, 1);
```

The second form can be used to set the three date fields simultaneously. For example to get a calendar for December 25 in the current year use

```
Calendar christmas = Calendar.getInstance();
christmas.set(christmas.get(Calendar.YEAR), Calendar.DECEMBER, 25);
```

### Example 4.28 (Using the Calendar class)

Try the following statements in BeanShell

```
bsh % import java.util.Date;
bsh % import java.util.Calendar;
bsh % Calendar now = Calendar.getInstance();
bsh % Date time = now.getTime();
bsh % print(time);
Sun Jun 15 10:04:12 EDT 2003
bsh % print(now.get(Calendar.YEAR));
2003
bsh % print(now.get(Calendar.MONTH)); // January is month 0
5
bsh % print(now.get(Calendar.DAY_OF_MONTH)); // June 15
15
bsh % print(now.get(Calendar.DAY_OF_WEEK)); // Sunday = 1
1
bsh % print(now.get(Calendar.DAY_OF_YEAR));
166
bsh % Calendar christmas = Calendar.getInstance();
bsh % int year = christmas.get(Calendar.YEAR);
bsh % christmas.set(year, Calendar.DECEMBER, 25);
bsh % print(christmas.getTime());
Thu Dec 25 11:13:01 EST 2003
```

This shows that the month numbers begin at 0 not 1 (June is month 5) and the days of the week begin at 1 (Sunday).  

### Example 4.29 (Leap years)

The following BeanShell statements

```
4.5 Other library classes

```java
import java.util.Calendar;

Calendar feb2003 = Calendar.getInstance();
feb2003.set(2003, Calendar.FEBRUARY, 1);

Calendar feb2004 = Calendar.getInstance();
feb2004.set(2004, Calendar.FEBRUARY, 1);

print(feb2003.getActualMaximum(Calendar.DAY_OF_MONTH)); // 28
print(feb2004.getActualMaximum(Calendar.DAY_OF_MONTH)); // 29
```

show that the `getActualMaximum` method properly accounts for leap years.

**Person class that uses Calendar**

We can easily use the `Calendar` class to determine how old a person is this year. Your employer could use it to find out when you should retire. Given that `birthYear` is the year of birth and `age` is the age this year in years the following statements calculate the age:

```java
Calendar now = Calendar.getInstance();
age = now.get(Calendar.YEAR) - birthYear;
```

Here is a simple class that uses these statements:

```java
package chapter4.calendar; // remove this line if you're not using packages
import java.util.Calendar;

/**
 * A simple class that represents a person by a name and year of birth
 * There is also a method to determine how old the person is this year.
 */
public class Person
{
    private String name;
    private int birthYear;

    /**
     * Construct object given name and year of birth.
     * @param name the name of the person
     * @param birthYear the year of birth
     */
    public Person(String name, int birthYear)
    {
        this.name = name;
        this.birthYear = birthYear;
    }
}
Classes, Objects, and Methods

/**
 * Return the name.
 * @return the name
 */
public String getName() {
    return name;
}

/**
 * Return the birth year
 * @return the birth year
 */
public int getBirthYear() {
    return birthYear;
}

/**
 * Return the age this year.
 * @return the age this year
 */
public int age() {
    Calendar now = Calendar.getInstance();
    return now.get(Calendar.YEAR) - birthYear;
}

Here we have not included the age as a private data field. We could have done this and calculated the age in the constructor. However, if the age is calculated in the constructor it will never change subsequently and there is a remote possibility that an object of this class exists for a few years and the age will be incorrect.

**Specialized Calendar class**

Some classes like Calendar are very complex and for specialized applications it might be useful to develop a simpler version of the class. This is called adapting a class and the simpler version is called an adapter class.

If we want to write a program that displays the calendar for a given month then we do not need the full complexity of the Calendar class:

1. The only parts of a date we need are the year and month.

2. We need the day of the week for the first day of the month. For example, the first day of the month might be a Thursday so we need to skip Sunday to Wednesday and number days from 1 beginning on Thursday.

3. We need to know the number of days in the month, properly accounting for February in a leap year.
4. We need the names of the months if we want to print headings for each month.

The Calendar class can easily be adapted for this purpose. To do this we develop a class called CalendarMonth that has the following public interface.

```java
public class CalendarMonth {
    public CalendarMonth() {...}
    public CalendarMonth(int year, int month) {...}
    public int getYear() {...}
    public int getMonth() {...}
    public int dayOfWeek() {...}
    public int daysInMonth() {...}
    public String monthName() {...}
    public String toString() {...}
}
```

Here the first constructor uses the year and month for today and the second one uses given values. The dayOfWeek method returns the day of the week for the first day of the month. The return value is in the range 1 (Sunday) to 7 (Saturday). The daysInMonth method returns the number of days in the month, properly accounting for February in a leap year. The monthName method returns one of the strings January to December and the toString method returns a string such as June 2003 which could be used to print headings.

To implement the class we need the private data field

```java
private Calendar calendar;
```

Now the constructors can simply create a Calendar object for the first day of the month and the methods can use methods from the Calendar class. Adapter classes are usually quite simple since all the work is being done by the adapted class using aggregation. Here is the complete class followed by an explanation of the constructors and methods.

```java
package chapter4.calendar; // remove this line if you're not using packages
import java.util.Calendar;
import java.text.SimpleDateFormat;

/**
 * A class that represents a month of a given year in a form suitable
 * for printing calendars.
 */
public class CalendarMonth {
    private Calendar calendar;

    public static final int JANUARY = Calendar.JANUARY;
```
public static final int FEBRUARY = Calendar.FEBRUARY;
public static final int MARCH = Calendar.MARCH;
public static final int APRIL = Calendar.APRIL;
public static final int MAY = Calendar.MAY;
public static final int JUNE = Calendar.JUNE;
public static final int JULY = Calendar.JULY;
public static final int AUGUST = Calendar.AUGUST;
public static final int SEPTEMBER = Calendar.SEPTEMBER;
public static final int OCTOBER = Calendar.OCTOBER;
public static final int NOVEMBER = Calendar.NOVEMBER;
public static final int DECEMBER = Calendar.DECEMBER;

public static final int SUNDAY = Calendar.SUNDAY;
public static final int MONDAY = Calendar.MONDAY;
public static final int TUESDAY = Calendar.TUESDAY;
public static final int WEDNESDAY = Calendar.WEDNESDAY;
public static final int THURSDAY = Calendar.THURSDAY;
public static final int FRIDAY = Calendar.FRIDAY;
public static final int SATURDAY = Calendar.SATURDAY;

/**
 * Construct a calendar for this month.
 */
public CalendarMonth()
{
    calendar = Calendar.getInstance();
    calendar.set(Calendar.DAY_OF_MONTH, 1);
}

/**
 * Construct a calendar for given year, month.
 * @param year the year
 * @param month the month in the range 0 to 11
 */
public CalendarMonth(int year, int month)
{
    calendar = Calendar.getInstance();
    calendar.set(year, month, 1); // months begin at 0
}

/**
 * Return the year for this calendar.
 * @return the year for this calendar.
 */
public int getYear()
{
    return calendar.get(Calendar.YEAR);
}

/**
 * Return the month for this calendar.
 * @return the month for this calendar.
 */
public int getMonth()
4.5 Other library classes

```java
/**
 * Return the day of week for the first day of this month.
 * @return the day of week for the first day of this month
 * in range 1 to 7 where 1 = Sunday, 7 = Saturday
 */
public int dayOfWeek()
{
    return calendar.get(Calendar.DAY_OF_WEEK);
}

/**
 * Return number of days in the month.
 * @return number of days in month.
 * Leap years are taken into account
 */
public int daysInMonth()
{
    return calendar.getActualMaximum(Calendar.DAY_OF_MONTH);
}

/**
 * Return the month name
 * @return the month name
 */
public String monthName()
{
    SimpleDateFormat f = new SimpleDateFormat("MMMM");
    return f.format(calendar.getTime());
}

/**
 * Return a string representation of a month
 * @return a string representation of a month
 */
public String toString()
{
    // example: June 2003
    SimpleDateFormat f = new SimpleDateFormat("MMMM yyyy");
    return f.format(calendar.getTime());
}
```
**Explanation of the class**

We define twelve static public constants for the months in the same range as the `Calendar` class and similarly seven constants for the day names. Since they are static constants there is only one set for all objects and they are referred to using names such as `CalendarMonth.JUNE`. Since the constants here are public we should have included a javadoc comment for each one of them.

For the default constructor we start with the date for today and set the day field to 1 for the first day of the month. Similarly, the second constructor uses the three argument form of the set method.

Since we set the calendar object to the first day of the month we simply use the expression

```java
    calendar.get(Calendar.DAY_OF_WEEK)
```

in the `dayOfWeek` method to return a number in the range 1 to 7 for the day of week for the first day of the month.

For the last day of the month we use the `getActualMaximum` method from Example 4.29 which returns the maximum value of any field so we apply it to the `DAY_OF_MONTH` field. The returned value properly accounts for leap years.

The month name can be obtained using a special format of `SimpleDateFormat` so we use it in the `monthName` method. Similarly we use another date format in the `toString` method.

**Testing the class**

There is not much testing to do since our class is just a specialized case of the `Calendar` class.

The following example shows how to test the class in BeanShell.

```bash
[EXAMPLE 4.30] (Using CalendarMonth in BeanShell)

    bsh % addClassPath("c:/book-projects/chapter4/calendar");
    bsh % CalendarMonth thisMonth = new CalendarMonth();
    bsh % print(thisMonth.dayOfWeek());
    1
    bsh % print(thisMonth.daysInMonth());
    30
    bsh % print(thisMonth.monthName());
    June
    bsh % print(thisMonth);
    June 2003
    bsh % CalendarMonth feb2004 = new CalendarMonth(2004, CalendarMonth.FEBRUARY);
    bsh % print(feb2004.dayOfWeek());
    1
    bsh % print(feb2004.daysInMonth());
    29
    bsh % print(feb2004.monthName());
    February
    bsh % print(feb2004);
    February 2004
```
4.5 Other library classes

In this test \texttt{thisMonth} refers to June, 2003.

To test the class using BlueJ follow the steps

1. Construct an object called \texttt{thisMonth} using the default constructor on the class menu (see Figure 4.9(a)).

2. Construct another object called \texttt{feb2004} using the other constructor on the class menu and enter 2004 for the year and \texttt{CalendarMonth.FEBRUARY} for the month. You now have two objects as shown in Figure 4.9(b).

3. Right click on the objects to get the object menu (see Figure 4.9(c)) and try the different methods.

4. If you use “inspect” on the object menu you will be able to see the static data fields by clicking on the “Show static fields” button in the Inspector window. These fields are also available by right clicking on the class rectangle.

Here is a short tester class that can also be used.

```
package chapter4.calendar; // remove this line if you’re not using packages
/**
 * A simple test program for the CalendarMonth class
 */
public class CalendarMonthTester
{
    public CalendarMonthTester()
    {
        ...
    }
```
public void doTest()
{
    CalendarMonth thisMonth = new CalendarMonth();
    System.out.println("First day of month is " + thisMonth.dayOfWeek());
    System.out.println("Number of days in month is " + thisMonth.daysInMonth());
    System.out.println("Month name is " + thisMonth.monthName());
    System.out.println("Calendar name is " + thisMonth);

    System.out.println();
    CalendarMonth feb2004 = new CalendarMonth(2004, CalendarMonth.FEBRUARY);
    System.out.println("First day of month is " + feb2004.dayOfWeek());
    System.out.println("Number of days in month is " + feb2004.daysInMonth());
    System.out.println("Month name is " + feb2004.monthName());
    System.out.println("Calendar name is " + feb2004);
}

4.5.2 Currency formatting

Another useful class in the java.text package is NumberFormat which can be used to format numbers as currency. The static method getCurrencyInstance constructs a currency format for your locale and returns a reference to it. Then you can use its format method to do the formatting. Therefore a currency formatter can be declared using

    NumberFormat currency = NumberFormat.getCurrencyInstance();

This is the same idea as in the Calendar class. A static method is used instead of a constructor to create an object. The method has no arguments: everyone will get the appropriate formatter for their locale and programs will be locale independent.

For a North American locale this formatter rounds numbers to two decimal places, puts a dollar sign at the beginning of the number, and uses a comma as the thousands separator.

**EXAMPLE 4.31** (Formatting currency) Try the BeanShell statements

```
bsh % import java.text.NumberFormat;
bsh % import java.util.Locale;
bsh % double salary = 100000.555;
bsh % NumberFormat currency = NumberFormat.getCurrencyInstance();
bsh % print(currency.format(salary));
$100,000.56
bsh % NumberFormat currencyCF =
    NumberFormat.getCurrencyInstance(Locale.CANADA_FRENCH);
bsh % print(currencyCF.format(salary));
100 000,56 $
```

The default currency object for the author corresponds to Locale.CANADA. Other locales such as Locale.CANADA_FRENCH can be specified. In this case the dollar sign is at the end, the decimal point is replaced by a period and spaces are used instead of commas to separate thousands.
4.5 Other library classes

4.5.3 Formatting fixed and floating point numbers (Java 1.4)

The material in this section is not so important in Java 5 since the static format method in the String class and the printf method in System.out can more easily be used to format numbers (see Section 4.2.5).

Another useful class in the java.text package is DecimalFormat which can be used to format numbers in fixed or floating point (scientific) format. In scientific notation numbers are usually represented in the form \( \pm m \times 10^e \) where \( 1 \leq m < 10 \) is called the mantissa and \( e \) is called the exponent. Fixed format numbers are rounded to a fixed number of digits after the decimal point without an exponent and are suitable for numbers in a small range such as 1 to 10.

For example, to specify fixed notation with 1 or more digits to the left of the decimal point and five digits to the right you can use the format

\[
\text{DecimalFormat fix } = \text{new DecimalFormat(" 0.00000;-0.00000");}
\]

Here the format string comes in two parts. To the left of the semi-colon is the format for non-negative numbers and to the right is the format for negative numbers. Here we specify a leading space for non-negative numbers and a minus sign for negative numbers. This means that in a column numbers of mixed sign will line up.

The format method has the prototype

\[
\text{public String format(double d)}
\]

so fix.format(d) would be used to format a double number as a string using the specified format. Here is a simple example.

\[\text{EXAMPLE 4.32 (Fixed format)}\] Try the statements

\[
\begin{align*}
\text{bsh } &\% \text{ import java.text.DecimalFormat;} \\
\text{bsh } &\% \text{ DecimalFormat fix } = \text{new DecimalFormat(" 0.00000;-0.00000");} \\
\text{bsh } &\% \text{ print(fix.format(Math.PI));} \\
&3.14159 \\
\text{bsh } &\% \text{ print(fix.format(-Math.PI));} \\
&-3.14159
\end{align*}
\]

The numbers are properly rounded.

Similarly, to specify a scientific format with one non-zero digit to the left of the decimal point, five digits after the decimal point and a three digit exponent we use the format

\[
\text{DecimalFormat sci } = \text{new DecimalFormat(" 0.00000E000;-0.00000E000");}
\]

Here is an example.

\[\text{EXAMPLE 4.33 (Scientific format)}\] Try the statements

\[
\begin{align*}
\text{bsh } &\% \text{ DecimalFormat sci } = \text{new DecimalFormat(" 0.00000E000;-0.00000E000");} \\
\text{bsh } &\% \text{ double d } = 1.2345678E-23; \\
\text{bsh } &\% \text{ print(sci.format(d));} \\
&1.23457E-023 \\
\text{bsh } &\% \text{ print(sci.format(-d));} \\
&-1.23457E-023
\end{align*}
\]
4.6 Review of OOP concepts

In this section we review and extend the basic OOP concepts of Chapter 3 and Chapter 4.

4.6.1 Constructing objects

In object-oriented programming the creation of objects is the most basic concept. Before an object can be used it must be constructed using a constructor call expression (Chapter 3, page 52) or by calling a static method in a class that returns an object of the class (factory method).

Using a constructor

Examples that use a constructor are

1. \( \text{Circle } c1 = \text{new Circle}(\text{new Point}(3,4), 5) \);
2. \( \text{Point } p = \text{new Point}(3,4) \);
3. \( \text{Point } q = \text{new Point}() \);
4. \( \text{Circle } c1 = \text{new Circle}(3,4,5) \);
5. \( \text{BankAccount } a = \text{new BankAccount}(123, \text{"Reginald Hill"}, 4000) \);
6. \( \text{SimpleDateFormat } f = \text{new SimpleDateFormat}(\text{"MMMM yyyy"}) \);

In each case the constructor call expression is underlined. In the first example there are two constructor call expressions, one is used as an argument in the other.

Corresponding to each constructor call expression is a constructor prototype. For the above examples the prototypes are

- \( \text{public Circle(Point p, double radius)} \)
- \( \text{public Point(double x, double y)} \)
- \( \text{public Circle(double x, double y, double radius)} \)
- \( \text{public BankAccount(int number, String name, double balance)} \)
- \( \text{public SimpleDateFormat(String pattern)} \)

A constructor prototype is the first line of the constructor declaration. It tells you how to write valid constructor call expressions. The general syntax for a constructor declaration was given in Chapter 3, Figure 3.28.

Using a static factory method

Examples of object construction that use a static method are

\( \text{Calendar } now = \text{Calendar.get Instance}(); \)
\( \text{NumberFormat } currency = \text{NumberFormat.getCurrencyInstance}(); \)
The **static method call expression** that constructs the object is underlined. Each such expression begins with the class name not an object name as in an instance method call expression (see Chapter 3, page 84). This particular kind of static method is often called a **factory method**, since its purpose is to manufacture a complex object such as a Calendar object.

**Using this as a constructor call expression**

We have seen two uses of this. It is used to access an instance data field. For example, in the Point class (page 119) we used `this.x` and `this.y` in the constructor to refer to the instance data fields because the constructor arguments had the same name.

Also, this can be used in a method call expression. For example, the QuadraticRootFinder class in Chapter 3, page 67 we used the method call expression `doCalculations()` in several places to aid in the calculations. We could have used `this.doCalculations()` to emphasize that the method is defined in “this” class although it was not necessary.

There is a third use of this which is quite useful in classes that contain several constructors. For example, in the Circle class we had three constructors

```java
public Circle()
{
    center = new Point();
    radius = 1;
}

public Circle(double x, double y, double r)
{
    center = new Point(x, y);
    radius = r;
}

public Circle(Point p, double r)
{
    center = p;
    radius = r;
}
```

The first two constructors are really special cases of the third one so we could have written them as

```java
public Circle()
{
    this(new Point(), 1);
}

public Circle(double x, double y, double r)
{
    this(new Point(x, y), r);
}
```
Here we are using `this` to call the third constructor. This does not seem like much of a simplification in this example, but it is in classes where constructor bodies contain lots of statements which would have to be duplicated.

Often there is a general constructor and all others are special cases. We can write the code once for the most general constructor and use `this` to refer to it in the other constructors.

**Default constructor**

In all the classes we have written we have included at least one constructor. It is common to see classes that have no constructors. For example in the `CircleTester` class (page 127) we included the no-arg constructor

```java
public CircleTester()
{
}
```

which does nothing since there are no data fields in this class. The purpose of the class is simply to provide the `doTest` method that can be executed in `BeanShell` or `BlueJ` as a simple test of the `Circle` class. In fact this constructor could have been completely omitted from the class and when you right click on the `CircleTester` class rectangle you will still see the no-arg constructor on the menu.

The reason is that if no constructor declarations at all are present in a class the compiler will automatically provide a so-called **default no-arg constructor**. This is often humorously called the Miranda convention:

> “You have the right to a constructor. If you do not have one, a default one will be provided for you by the compiler.”

The no-arg constructor, whether we provide it or let the compiler provide it, allocates memory for an object, like any constructor, but it also provides default initialization for any uninitialized data fields according to the following rules:

1. A value of 0 is assigned to all uninitialized numeric data fields.

2. A reference value of `null` is assigned to all uninitialized data fields of object type (see Section 4.6.2 below).

You can try it with `BlueJ` by writing the following three very simple classes in a project.

```java
public class One
{
    // the simplest of all classes
}

public class Two
{
    private int k;
    private One one;
```
public class Three
{
    private int k;
    private One one;

    public Three()
    {
    }
}

Construct an object of each type and use “Inspect” on the object menu to see the data fields for objects of classes Two and Three. In either case you will see

private int k = 0;
private One one = null;

indicating that there is a default initialization.

Of course, it is best not to rely on default initialization and a better version of Two or Three would be

public class Four
{
    private int k;
    private One one;

    public Four()
    {
        k = 0;
        one = null;
    }
}

Here the initialization is explicit.

### 4.6.2 Object references

Constructing an object is a three step process for the Java run-time system.

1. Memory space is allocated for the object and its instance data fields.

2. A reference to the object is returned so that it can be located.

3. This reference is assigned as the value of an object reference variable.

Thus, an object is located using an **object reference**. Each object reference can be stored in a variable called an **object reference variable** whose value gives the location of the object. The name of this variable is called the object name.
A picture of an object and a reference to it is shown in Figure 4.10(a). This is often called the **box and arrow** notation. The object and its data fields are shown as a rounded box. The reference variable is shown as a square box with an arrow pointing to the object to symbolize the reference to the object, and the picture corresponds to the statement

\[
\text{ClassName objectName} = \text{new ClassName (actualArguments)};
\]

The right side is the constructor call expression. When it is executed the memory space for the object is obtained and a reference to the object is returned. This reference is then assigned as the value of the object reference variable \( \text{objectName} \). For example,

\[
\text{Circle c} = \text{new Circle(3, 4, 5)};
\]

Object types are quite different than the primitive types such as \text{int} and \text{double}. This can be seen by comparing Figure 4.10 with Figure 2.1 and Figure 2.2. However the reference variable itself acts like a primitive type.

**Null references**

It is common to declare the name of an object without constructing the object right away. This situation is shown in Figure 4.10(b). Here we have no arrow. This corresponds to the declaration

\[
\text{ClassName objectName};
\]

A reference that does not point to an object yet is called a **null reference** and is indicated by the Java keyword \text{null} so this declaration can be written as

\[
\text{ClassName objectName} = \text{null};
\]

These two declarations are not quite the same. In the first case we have an uninitialized variable. In the second case the variable is initialized to the value \text{null}.

For example,

\[
\text{Circle c} = \text{null};
\]

indicates that \( \text{c} \) is an object reference variable for an object of type \text{Circle} but it does not yet refer to any object. For such a variable it is a run-time error to invoke methods on a null reference.

For example, the statements

![Box and arrow representation of an object and a reference to it.](image-url)
Circle c = null;
double radius = c.getRadius();

would give an error called a NullPointerException (references are sometimes called pointers) because there is no object to invoke the getRadius method on. This is a common error that you must watch out for.

**Comparison of primitive and reference types**

We have encountered two types of variables in Java:

**primitive types**
- numeric types such as int, long, char, float, and double introduced in Chapter 2 and the boolean type which will be introduced in a later Chapter.

**reference types**
- object types such as String, Circle, and BankAccount introduced in Chapter 3 and Chapter 4.

These two types work in very different ways. For example, if d is a double variable then we cannot take its square root by doing something like d.sqrt(). Since d is not an object it has no methods. Instead we need to apply the static Math.sqrt method and use Math.sqrt(d). But if c is a reference to a Circle object then we can use c.getCenter() to invoke the getCenter method on c and return the center of the circle. Thus, you cannot invoke methods on variables of primitive type.

The situation is shown in Figure 4.11 Here we see in part (a) that the value of the double variable area is stored directly in the memory allocated for area but the value of greeting is a reference to the string object "Hello Fred" indicated by the arrow, and the reference is stored in the memory allocated for greeting. The String object, namely the string value "Hello Fred", is stored elsewhere in memory reserved for the object. In fact as mentioned above we can think of the reference greeting as a primitive type – the memory address or location of the object.

![Figure 4.11: Comparison of primitive and reference types.](image-url)
Why do we need both primitive and reference types?

Why not use reference types for all variables? This could have been done in Java and is done in other OOP languages such as Smalltalk. However, some types such as `int` and `double` were specified as primitive types to make their use more efficient as there is a certain amount of overhead in following a reference to find its object.

Assignment statements for reference types

Another major difference between primitive types and reference types occurs with the interpretation of assignment statements such as

```
a = b;
```

If `a` and `b` are variables of primitive types then we know this statement means to assign the value of `b` as the value of `a`. In other words, `a` and `b` have the same values after the assignment. This is shown in Figure 4.12. In part (a), before the assignment `a = b`, variable `a` has the value 17, and variable `b` has the value 19. In part (b), after the assignment `a = b`, both variables have the same value 19.

However, if `a` and `b` are references to objects then the situation is quite different, as shown in Figure 4.13. In part (a), before the assignment `a = b`, there are separate references for each of the objects `objectA` and `objectB` so `a` and `b` hold different values. In part (b), after the assignment `a = b`, `a` and `b` now hold the same values, namely a reference to `objectB`.

Unless there was some other reference to `objectA`, it has now become an orphan. This is indicated by the question mark beside `objectA`. The Java interpreter contains a garbage collector that will eventually delete it and reclaim the memory it uses.

**Example 4.34** (BankAccount example of `a = b`) For the `BankAccount` class the BeanShell statements

```bash
bsh % addClassPath("c:/book-projects/chapter4/bank_account");
bsh % BankAccount fred = new BankAccount(123, "Fred", 150.0);
bsh % BankAccount mary = new BankAccount(345, "Mary", 350.0);
bsh % mary = fred;
bsh % mary.withdraw(100.0);
```
4.6 Review of OOP concepts

4.6.3 Using references as arguments and method return values

We have seen in Section 4.4.1 (page 117) on association that object references of one class can be used in another class in the following ways.

1. **local variables in the body of a constructor or method**
   For example, on page 118 we use
   
   ```java
   TriangleCalculator tri = new TriangleCalculator(a,b,g);
   ```
   
   in the body of the `doTest` method in `TriangleCalculatorTester`.

2. **constructor or method arguments**
   For example, in `Circle` (page 122) we have the constructor prototype
   
   ```java
   public Circle(Point p, double r)
   ```
   
   so the first argument is a reference to a `Point`.

3. **method return values**
   For example, in most of our classes we have used the `toString` method which returns a reference to a `String` object. As another example, the `getCenter` method in the `Circle` class contains the `return` statement
return center;

which returns a reference to the Point object for the center of the circle.

4. **instance data fields**
   This is aggregation and we have seen several examples. For example, PasswordGenerator (page 113) has the instance data field
   ```java
   private Random random;
   ```
   and in CalendarMonth (page 133) we have the instance data field
   ```java
   private Calendar calendar;
   ```

### 4.6.4 Data encapsulation and integrity

One of the benefits of object oriented programming is that data can be spread out over a number of objects. Each object is then responsible for its own data, independent of the data of any other object. This is called **data encapsulation**.

For example, in the Point class each object has its own copies of the data fields x and y representing the coordinates of the point. This particular class was designed to be immutable: it is not possible to change the coordinates of a Point object after it was constructed since there are no “set” methods and the data fields are private.

If possible it is a good idea to make your classes immutable. This is not possible in general. Many classes, such as BankAccount (page 106), must be mutable since the withdraw and deposit methods must change the account balance.

Having private data fields is a necessary condition for the immutability of a class. For example, in the Point class we could violate immutability in two ways:

- **make the data fields public instead of private**
  ```java
  public double x;
  public double y;
  ```

- **provide setX and setY methods**:
  ```java
  public void setX(double x)
  {
    this.x = x;
  }

  public void setY(double y)
  {
    this.y = y;
  }
  ```
With public data fields we can construct an object and then directly change its data fields as follows:

```java
Point = new Point(3,4);
p.x = 4; // 4 is new x coordinate
p.y = 5; // 5 is new y coordinate
```

If you try to do this with private data fields the compiler will give you an error message. Similarly, if the “set” methods are included but the data fields are still private then the data fields can be changed indirectly as follows:

```java
Point = new Point(3,4);
p.setX(4); // 4 is new x coordinate
p.setY(5); // 5 is new y coordinate
```

Later we will see that if we need to make the class mutable the “set” methods are preferable to making the data fields public.

The same ideas apply to the BankAccount class. To maintain data integrity it would be necessary to ensure that an account cannot have a negative balance, and that the withdraw method cannot withdraw more money than is available. Our version of the class does not ensure these conditions but we will be able to modify the class when we learn about conditional statements.

### Side-effects

When a class is associated with a mutable class it is possible to have undesirable side-effects that violate data encapsulation. To see this we will write a new mutable version of the Point class called MPoint that contains setX and setY methods. The class is in a BlueJ project called book-projects/chapter4/side_effects.

```java
package chapter4.side_effects; // remove this line if you’re not using packages

/**
 * This class is like Point except it is a mutable version
 * with setX and setY methods. We have also added a copy constructor.
 */
public class MPoint
{
    private double x;
    private double y;

    /**
     * Construct a point from its coordinates.
     * @param x the x coordinate of the point
     * @param y the y coordinate of the point
     */
```

3 We have also added a copy constructor to be discussed below.
public MPoint(double x, double y) {
    this.x = x;
    this.y = y;
}

/**
 * Construct the default point (0,0).
 */
public MPoint() {
    x = 0.0;
    y = 0.0;
}

/**
 * Copy constructor
 * @param p point to copy
 */
public MPoint(MPoint p) {
    x = p.x;
    y = p.y;
}

/**
 * Return the x coordinate of this point.
 * @return the x coordinate of this point
 */
public double getX() {
    return x;
}

/**
 * Return the y coordinate of this point.
 * @return the y coordinate of this point
 */
public double getY() {
    return y;
}

/**
 * Set a new value for the x coordinate
 * @param x the new x coordinate
 */
public void setX(double x) {
    this.x = x;
}
4.6 Review of OOP concepts

/**
 * Set a new value for the y coordinate
 * @param y the new y coordinate
 */
public void setY(double y)
{
    this.y = y;
}

/**
 * Return a string representation of an MPoint object.
 * @return a string representation of an MPoint object
 */
public String toString()
{
    return "MPoint[" + x + ", " + y + "]";
}

In the same project we also write the following version of the Circle class called MCircle that uses MPoint instead of Point and violates data encapsulation.

### Class MCircle

```java
package chapter4.side_effects; // remove this line if you're not using packages

/**
 * This class is identical to Circle except it used MPoint, the mutable point
 * class, instead of Circle, which uses the immutable Point class.
 */
public class MCircle
{
    private MPoint center;
    private double radius;

    /**
     * Construct circle with given center point and radius.
     * @param p the center of the circle
     * @param r the radius of the circle
     */
    public MCircle(MPoint p, double r)
    {
        center = p;
        radius = r;
    }

    /**
     * Construct circle with given center coordinates and radius.
     * @param x the x coordinate of the circle center
     * @param y the y coordinate of the circle center
     */
    public MCircle(double x, double y)
    {
        center = new MPoint(x, y);
        radius = r;
    }

    // other methods...
}
```

```java
public class MCircle
{
    private MPoint center;
    private double radius;

    /**
     * Construct circle with given center point and radius.
     * @param p the center of the circle
     * @param r the radius of the circle
     */
    public MCircle(MPoint p, double r)
    {
        center = p;
        radius = r;
    }

    // other methods...
}
```
Classes, Objects, and Methods

```
*/
public MCircle(double x, double y, double r)
{
    center = new MPoint(x,y);
    radius = r;
}
/**
 * Construct a default circle: a unit circle with center (0,0)
 * and radius 1.
 */
public MCircle()
{
    center = new MPoint();
    radius = 1;
}
/**
 * Return radius of circle.
 * @return radius of circle
 */
public double getRadius()
{
    return radius;
}
/**
 * Return center of circle.
 * @return center of circle
 */
public MPoint getCenter()
{
    return center;
}
/**
 * Return a string representation of an MCircle object.
 * @return a string representation of an MCircle object
 */
public String toString()
{
    return "MCircle[" + center + ", " + radius + "]";
}
}
```

The following example illustrates the side-effects.

**Example 4.35** (An undesirable side-effect) Try the statements

```
bsh % addClassPath("c:/book-projects/chapter4/side_effects");
bsh % MPoint p = new MPoint(3,4);
```
and notice that the change of the $x$ coordinate of $p$ from 3 to 999 has miraculously changed the $x$ coordinate of the center of the circle to 999 too.

Why should a change in a point that should have nothing to do with the circle, change the circle object? The reason is that $p$ and the instance data field $center$ both reference the same object since the constructor contains the statement

```java
center = p;
```

so any change in this object through $p$ is also a change in the center of the circle.

You can also confirm these results using BlueJ as follows

1. Construct an $MPoint$ object called $p$ with coordinates 3 and 4.

2. Construct an $MCircle$ object called $c$ using the constructor that has an $MPoint$ object and choose $p$ as the $MPoint$ object and a radius of 5.

3. Now use the $setX$ method on the object menu of $p$ and change the $x$ coordinate to 999.

4. Finally, use the inspector choice on object menu for $c$ and examine the object reference $center$ to see that the $x$ coordinate of the center is now 999.

The following example shows another way to violate data encapsulation with $MCircle$.

**Example 4.36** (Another side-effect) Try the statements

```java
bsh % addClassPath("c:/book-projects/chapter4/side_effects");
bsh % MCircle c = new MCircle(3,4,5);
bsh % print(c);
MCircle[MPoint[3.0, 4.0], 5.0]
bsh % p.setX(999);
bsh % print(c);
MCircle[MPoint[999.0, 4.0], 5.0]
```

and notice that we can use $getCenter$ to obtain a reference to the center point. Then we can use this reference to change the $x$ coordinate from 3 to 999. The reason here is the statement

```java
return center;
```

in the $getCenter$ method, which also returns a reference to $p$. 

```java
bsh %
```
These situations involving undesirable side-effects are shown in Figure 4.14. Here it is clear that there is only one MPoint object and it can be modified either through reference \( p \) or reference \( \text{center} \).

\[
\text{Figure 4.14: A side-effect after } p \text{ changes the } x \text{ from } 3 \text{ to } 999.
\]

To see that this side-effect does not occur in the original versions of these classes try the following example in BeanShell.

**Example 4.37 (No side-effect in original classes)** Try the statements

```java
bsh % addClassPath("c:/book-projects/chapter4/geometry");
bsh % import Point; // necessary or we get java.awt.Point
bsh % Point p = new Point(3,4);
bsh % Circle c = new Circle(p, 5);
bsh % print(c);
Circle[Point[3.0, 4.0], 5.0]
bsh % p = new Point(999,4);
bsh % print(c);
Circle[Point[3.0, 4.0], 5.0]
```

and notice now that there is no change to the center coordinates of the circle. This occurs because Point is an immutable class so there is no way to change the Point object that was created. All we can do is create a new object for \( p \) to reference in the statement

\[
p = \text{new Point}(999,4);
\]

So now instead of having two references \( p \) and \( \text{center} \) to the same object we have references to different objects. Now \( \text{center} \) is a reference to \((3,4)\) and the \( p \) is a reference to \((999,4)\).
4.6 Review of OOP concepts

The same analysis applies to the String class since it is also immutable. As a general rule it is best to make classes immutable if possible.

Before fixing MCircle to remove side effects we discuss the concept of a copy constructor.

Copy constructor

A copy constructor makes a copy of the object referenced by its argument. For a class called ClassName the copy constructor has the prototype

    public ClassName (ClassName objectName)

We have included the copy constructor

    public MPoint (MPoint p)
    {
        x = p.x
        y = p.y
    }

in the MPoint class. It is legal to use p.x and p.y here even though the data fields are private because we are inside the MPoint class itself and instance data fields can be accessed directly in this case.

Using the copy constructor we can modify the MCircle class so that side-effects do not occur. The first modification is to change the constructor that accepts an MPoint reference as an argument to

    public MCircle (MPoint p, double r)
    {
        center = new MPoint (p);
        radius = r;
    }

This makes a copy of the MPoint object referenced by p and assigns its reference to center. Now the MCircle object has its own copy of this object which is not affected by any changes to the callers object referenced by p.

The second modification is to change the getCenter method to

    public MPoint getCenter()
    {
        return new MPoint (center);
    }

Again, this returns a reference to a copy of the center object not a reference to the center object itself. In other words, by making copies we do not give away to the caller a reference to our private data field of type MPoint.

These changes are tested in a project called book-projects/chapter4/no_side_effects that contains our MPoint class and the following version of MCircle.
package chapter4.no_side_effects; // remove this line if you’re not using packages

/**
 * This class is like the original MCircle class
 * but we have made it immutable so there are no side-effects.
 */
public class MCircle {
    private MPoint center;
    private double radius;

    /**
     * Construct circle with given center point and radius.
     * @param p the center of the circle
     * @param r the radius of the circle
     */
    public MCircle(MPoint p, double r) {
        center = new MPoint(p);
        radius = r;
    }

    /**
     * Construct circle with given center coordinates and radius.
     * @param x the x coordinate of the circle center
     * @param y the y coordinate of the circle center
     * @param r the radius of the circle
     */
    public MCircle(double x, double y, double r) {
        center = new MPoint(x, y);
        radius = r;
    }

    /**
     * Construct a default circle: a unit circle with center (0,0)
     * and radius 1.
     */
    public MCircle() {
        center = new MPoint();
        radius = 1;
    }

    /**
     * Return radius of circle.
     * @return radius of circle
     */
    public double getRadius()
Now run the BeanShell test in Example 4.35 using this version of MCircle:

**EXAMPLE 4.38** (The side-effects are gone) Try the statements

```java
bsh % addClassPath("c:/book-projects/chapter4/no_side_effects");
bsh % MPoint p = new MPoint(3,4);
bsh % MCircle c = new MCircle(p, 5);
bsh % print(c);
MCircle[MPoint[3.0, 4.0], 5.0]
bsh % p.setX(999);
bsh % print(c);
MCircle[MPoint[3.0, 4.0], 5.0]
```

and the side-effects are gone: changing the x coordinate does not change the center of the circle. ■

With the immutable version of MCircle we have the situation shown in Figure 4.15. Now there are two MPoint objects instead of one. Side-effects like this always arise from mutable classes and references and cannot occur for primitive types such as int or double.

**BankAccount example**

As a simple example of association and aggregation that has desirable side-effects consider a class called TransferAgent whose objects can take two BankAccount and transfer a given amount of money from one account to the other. The public class interface is given by

```java
public class TransferAgent
{
```
Figure 4.15: The side-effect disappears after \( p \) changes \( x \) from 3 to 999.

public TransferAgent(BankAccount from, BankAccount to) {...}
public void transfer(double amount) {...}
}

Here the constructor arguments specify references to the two accounts and the \texttt{transfer} method does the transfer of a specified amount from the \texttt{from} account to the \texttt{to} account. Therefore the \texttt{TransferAgent} is associated with (or uses) the \texttt{BankAccount} class.

The class implementation needs two private data fields of type \texttt{BankAccount} to hold references to the two accounts that are involved in the transfer. Therefore the class also illustrates aggregation. The constructor will initialize these fields. The \texttt{transfer} method just needs to withdraw the given amount from one account and deposit it in the other account using the \texttt{withdraw} and \texttt{deposit} methods. Here is the complete class.

```java
package chapter4.bank_account; // remove this line if you're not using packages
/**
 * A TransferAgent object can be used to transfer a given amount of money
 * from one account to another.
 */
public class TransferAgent {
    private BankAccount from;
    private BankAccount to;

    /**
     * Construct an object for two accounts
     * @param from the "from" account
     */
```
4.6 Review of OOP concepts

```java
public TransferAgent(BankAccount from, BankAccount to) {
    this.from = from;
    this.to = to;
}

/**
 * Transfer the given amount of money from one account to the other.
 * @param amount amount to transfer
 */
public void transfer(double amount) {
    from.withdraw(amount);
    to.deposit(amount);
}
```

There are side-effects since the `BankAccount` class is mutable and the `transfer` method will change the balance data field of the two `BankAccount` objects given as arguments in the constructor. However, here the account objects are changed from inside the class, by the `transfer` method, not from outside it as in the `MCircle` Example 4.35, where the center of the circle was changed from outside the class. In fact the purpose of the `transfer` method is to produce these side-effects. You can try this class in `BeanShell` or `BlueJ`.

**Example 4.39 (Desirable side-effects)** The statements

```bash
bsh % addClassPath("c:/book-projects/chapter4/bank_account");
bsh % BankAccount fred = new BankAccount(123, "Fred", 1000);
bsh % BankAccount mary = new BankAccount(345, "Mary", 1000);
bsh % TransferAgent agent = new TransferAgent(fred, mary);
bsh % agent.transfer(500);
bsh % print(fred);
BankAccount[number=123, name=Fred, balance=500.0]
bsh % print(mary);
BankAccount[number=345, name=Mary, balance=1500.0]
```

transfer $500 from the `fred` account to the `mary` account.

In `BlueJ` try the following steps.

1. Create two `BankAccount` objects with the account numbers, names, and initial balances given above in the `BeanShell` example.

2. Create a `TransferAgent` object called `agent` for the two accounts as given in the above `BeanShell` example. The result is shown in Figure 4.16

3. From the `agent` object menu choose `transfer` and enter 500 as the amount.

4. Inspect the two account objects or use their `getBalance()` methods to verify that the transfer was made.
4.6.5 Instance variables and methods

We now summarize the concepts related to instance variables and methods.

**Instance variables**

The instance variables (instance data fields) of a class, if any, are declared in the class but outside any method or constructor. Each object from the class has its own set of these instance variables. Each instance method has access to these variables to examine or change the state of the object.

**Instance methods**

The methods we have written so far have been methods that we can invoke on an object. We have called these methods *instance methods* or *object methods*. In BlueJ they are accessed by right clicking on an object and choosing a method from the object menu.

In Java every method belongs to some class and when we construct an object from a class we often say that the object is an instance of the class. Each class we use or write usually has one or more instance methods. These methods define the behavior of objects from the class. A template for a method declaration was given in Chapter 3, Figure 3.29.

**Using an instance method**

To use a method means to invoke it on an object. For example, the `getArea` method in the `CircleCalculator` class in Chapter 3 (page 106), returns the area of the circle so we say that a `CircleCalculator` object knows how to compute the area of a circle – this is part of its behavior.

When an instance method is executed it is associated with some object. For example, if `circle` is a `CircleCalculator` object then the statement

```java
double area = circle.getArea();
```
4.6 Review of OOP concepts

String letter = LETTERS.substring(index, index + 1);

public String substring(int first, int lastPlusOne) {
    ...
}

Figure 4.17: Correspondence between call expressions and prototypes

calculates the area and assigns it to variable area. The expression circle.getArea() is called an **instance method call expression**, and we say that we are invoking the getArea method on the circle object. We can also say that we are **sending a message** to an object.

Here are some examples we have used in Chapter 4 with each instance method call expression underlined.

1. int length = name.length();
2. char first = test.toUpperCase().charAt(0);
3. password = password + LETTERS.substring(index, index + 1);
4. String upper = test.toUpperCase();
5. account.deposit(100);
6. String firstInitial = firstName.substring(0,1).toUpperCase();
7. double x = c2.getCenter().getX();
8. doCalculations(); // this is the implied object
9. long t = now.getTime();
10. String n3 = f3.format(now);

There is a direct correspondence between a method call expression and the method prototype as shown by the example in Figure 4.17.

In the simplest case an instance method call expression has one of the forms (see Chapter 3, page 84)

-objectName.methodName (actualArguments)
-this.methodName (actualArguments)

The second form refers to a method in its own class and the “this.” part can be omitted.

Examples (2), (6), and (7) illustrate an important idea called **message composition**. For example, in (2) we first send the toUpperCase message to the string test. This returns another string and we send the charAt(0) message to it to finally obtain an upper case version of the first character of the string test.

Similarly, in (7) we send the getCenter() message to the point c2. This returns a point and we send the getX() method to it to finally obtain the x coordinate of the center of the circle.
Method composition

Instance message composition can be extended to several steps so the general form is

\[
\text{objectName.methodName(args1)} . \text{methodName(args2)} ... \text{methodNameN(argsN)}
\]

This means that the expression can be extended to several steps so the general form is

\[
\text{this.methodName(args1)} . \text{methodName(args2)} ... \text{methodName(argsN)}
\]

As mentioned above “\text{this.”}” can be omitted. Read these expression from left to right and as long as each expression returns the right kind of object the entire expression will make sense.

4.6.6 Static variables, constants, and methods

Static variables

Variables can be declared static using the static modifier on the variable declaration. We have not used static variables yet but an example we will use below is

\[
\text{private static int count = 0;}
\]

which declares \text{count} to be a static variable whose initial value is 0.

A static variable is quite different from an instance variable where each object has its own copy of the variable. This is shown in Figure 4.15 where the two \text{MPoint} objects clearly have their own copies of the \text{x} and \text{y} data fields. For a static variable there is only one copy and all objects of the class share this variable – any constructor or method in the class can inspect or modify it. In the above example any change in the value of \text{count} made by some object will also be seen by all other objects from the class. We say that \text{count} is a shared variable.

Static constants

A static constant in a class is declared in the same way as a static variable but the final keyword is included. Constants in a class are normally declared to be static since there is no need for each object to have its own copy of a constant. The constant \text{Math.PI} is an example from the \text{Math} class and we have also introduced some examples of our own. In the \text{PasswordGenerator class} (page 113) we defined a string of lower case letters as a static constant:

\[
\text{public static final String LETTERS = "abcdefghijklmnopqrstuvwxyz";}
\]

and in the \text{CalendarMonth class} we used static integer constants such as

\[
\text{public static final int JANUARY = Calendar.JANUARY;}
\]

Static methods

The static modifier is used on a method prototype and declaration to distinguish a static method from an instance method. We have not written a static method yet but we have used some. A static method in a class is not associated with any object of the class. Thus, the statements in the body of a static method cannot refer to any of the instance data fields.
Using static methods

The syntax for using a static method is similar to that of an instance method except the class name is used in place of the object name. For example, all the methods in the `Math` class are static methods (see Chapter 2) so to calculate $\sqrt{2}$ and assign it to a variable we would use

```java
double root2 = Math.sqrt(2.0);
```

The expression `Math.sqrt(2.0)` is called a **static method call expression**. The class name `Math` must be present so the compiler will know which class contains the `sqrt` method. The `Math` class is an example of a class that contains only static methods.

Here are some examples we have used in Chapter 3 and Chapter 4 with each static method call expression underlined.

1. `beta = Math.toDegrees(beta);`
2. `System.out.println("Area: " + area);`
3. `Calendar now = Calendar.getInstance();`
4. `NumberFormat currency = NumberFormat.getCurrencyInstance();`

Examples like (2) are interesting. Here `System` is the name of a class so `System.out` is a static variable in this class. The type of this variable is `PrintWriter` which has several `print` and `println` instance methods.

The methods in the static method call expressions in (3) and (4) are called **factory methods** because they are static methods in a class whose job it is to construct an object from some class and return a reference to it.

Thus, the general syntax for invoking a static method is

```
ClassName .methodName (actualArguments)
methodName (actualArguments)
```

The second form would correspond to a static method in its own class.

Counting the number of objects created from a class

For the `Point` class (page 119) suppose that we want to count how many objects from the class have been instantiated (created). We can do this with a static variable and a static method using the project called `book-projects/chapter4/static_variable` that contains a modified version of the `Point` class from project `book-projects/chapter4/geometry`. The new `Point` class is obtained by adding the following static variable to the data field section (outside any constructor or method).

```java
private static int count = 0;
```

Now add the following statement to the body of each constructor;

```java
    count++;
```

Thus, each time an object is created `count` will be incremented.

Finally, add the following static method to the class
Classes, Objects, and Methods

public static int getCount()
{
    return count;
}

Testing the class with BeanShell  Now you can test the class in BeanShell as follows.

**Example 4.40** (Counting the number of objects) The statements

```java
bsh % addClassPath("c:/book-projects/chapter4/static_variable");
bsh % import Point; // necessary of we get java.awt.Point
bsh % print(Point.getCount());
0
bsh % Point p1 = new Point(1,2);
bsh % Point p2 = new Point(3,4);
bsh % Point p3 = new Point(4,5);
bsh % Point p4 = p1;
bsh % int objectCount = Point.getCount();
bsh % print(objectCount);
3
```

show how to use the static method to count the number of Point objects that have been created. If you were expecting 4 as an answer instead of 3 recall from Figure 4.13, page 147, that p1 and p4 are references to the same object, the Point object with coordinates (1,2).

Testing the class with BlueJ  You can also use BlueJ to do the same test:

1. Create a few Point objects.
2. Now you will not find the getCount method on the object menu since it is not an instance method. Instead right click on the Point class box and it will be there.
3. Call the getCount method to display the number of objects created and displayed so far.

A class can contain a mixture of instance methods and static methods or it can contain only methods of one kind. This is often a design decision. One possible advantage of a static method is that it is not necessary to first construct an object before using the method. Only the class name is needed. Of course, to execute an instance method it is always necessary to construct an object first.

The Math class is an example of a class containing only static methods. If the designer had used instance methods then it would always be necessary to construct some “math” object before using one of the math functions and this does not make sense.

Some classes have both kinds of methods. For example, the NumberFormat class (see Section 4.5.2) has the static method getCurrencyInstance but it also has the instance method format.

---

4The String class is an example
The classes we have written so far have been classes containing only instance methods. If an instance method does not access any instance data fields then we could choose to make it an instance method or a static method. For example, the doTest method in the simple classes TriangleCalculatorTester, page 118, CircleTester, page 127, and CalendarMonthTester, page 137, do not access any data fields. In fact, these classes have no data fields. Therefore these methods could have been made static. The only difference in BlueJ is that we would now access the doTest() method from the class menu instead of the object menu.

4.6.7 Kinds of variables and arguments

So far we have seen four kinds of variables in Java.

- **Instance data fields**: They are instance variables that are declared outside any constructor or method in a class and can be accessed directly by any instance method in the class.

- **Static data fields**: They are static variables that are declared outside any constructor or method in a class and can be accessed by any instance or static method in the class.

- **Local variables**: They are variables declared in a method or constructor body and are only accessible within this body.

- **formal arguments**: They are declared in the constructor or method prototype and become local variables inside the constructor or method body.

4.6.8 Call by value argument passing mechanism

When a method or constructor call expression is executed, it is important to understand how the actual arguments in the expression are assigned to the formal arguments specified in the method or constructor declaration.

For example, in Figure 4.3 (page 105) there are three formal arguments that need to be matched with the three actual arguments in the constructor call expression. When the constructor is called the values of the actual arguments are assigned as the values of local variables with the names accountNumber, ownerName and initialBalance.

Thus, in the body of the constructor, accountNumber will have the value 123, ownerName will have the value of a reference to the String object "Peter Pascoe", and initialBalance will have the value 125.50.

This is the call by value argument passing mechanism. The rules are simple:

- If an actual argument is a variable then a copy of its value is supplied as the value of the formal argument.

- If an actual argument is a literal (literal string or number for example) then this literal value is supplied as the value of the formal argument.

- If an actual argument is an expression then the expression is evaluated and its value is supplied as the value of the formal argument.
Call by value for an argument of primitive type

A consequence of call by value is that if a constructor or method argument is a primitive type and a variable is used as an actual argument, then the value of that variable can not be changed by the method since only a copy of the variable’s value, not its location, is passed to the method. For example consider the following method, which could be a static or an instance method since it doesn’t refer to the instance data fields of any class.

```java
void addOne(int k)
{
    k = k + 1;
}
```

If the intent of this method is to add one to variable k, and somehow return it to the caller of the method, it does not work since k is a local variable which receives only a copy of the caller’s value, not its location.

An interesting feature of BeanShell is that we can define this method without the keywords static or public and BeanShell will treat it as static method without having to specify to which class it belongs. Here is an example.

**[EXAMPLE 4.41]** (Arguments of primitive type) The BeanShell statements

```bash
bsh % void addOne(int k) { k = k + 1; }
bsh % int m = 5;
bsh % addOne(m);
bsh % print(m);
   5
bsh % print(k);
   // Error: Undefined variable or class name, parameter: arg to method: print : at Line: 6 : in file: <unknown file> : print ( k )
```

do not change the value of m since the value of m, not its location, is assigned to k in the method body and 1 is added to the value of this local variable not to the callers variable m. When the method finishes executing the local variable k disappears and is no longer available. This feature of BeanShell is very convenient for testing static methods.

In BlueJ: we can write the small tester class

```java
class ArgumentTester1
{

```

package chapter4.arguments; // remove this line if you’re not using packages

```java
/**
 * Illustrating call by value for primitive types
 */
public class ArgumentTester1
{  
```
public void doTest()
{
    int m = 5;
    addOne(m);
    System.out.println(m);
}

private static void addOne(int k)
{
    k = k + 1;
}
}

Now we can construct a ArgumentTester1 object and invoke its doTest() method, which will call the static addOne method and display the resulting value of m which is still 5.

Call by value for an argument of reference type (object type)

For reference types the situation is quite different. We still use “call by value” but now the value passed is a copy of a reference to the caller’s object so there will now be two references to the caller’s object. Consider the following method.

```java
void addOneDollar(BankAccount b)
{
    b.deposit(1);
}
```

Here b is a reference to a BankAccount object and this class is mutable, so the deposit statement will change the balance of the BankAccount object referenced by b. The following BeanShell example shows this. We have included a copy of BankAccount in the arguments project.

**Example 4.42 (Arguments of reference type)** The BeanShell statements

```bash
bsh % addClassPath("c:/book-projects/chapter4/arguments");
bsh % void addOneDollar(BankAccount b) { b.deposit(1); }
bsh % BankAccount a = new BankAccount(123, "Fred", 1000);
bsh % addOneDollar(a);
bsh % print(a);
BankAccount[number=123, name=Fred, balance=1001.0]
```

show that the balance of the object referenced by a has $1 added to it because inside the method b is also a reference to this object.

This situation is illustrated in Figure 4.18 Before the addOneDollar method call, part (a), a is the only reference to the object. During the method call, part (b), b is a local variable that is a copy of a so it refers to the same object. After the method call b disappears and we have the original situation but with the balance increased by one dollar.

In BlueJ: we can write the small tester class
Classes, Objects, and Methods

Figure 4.18: Call by value using references.

```java
package chapter4.arguments; // remove this line if you're not using packages
import chapter4.bank_account.BankAccount; // remove this line if you're not using packages

/**
 * Illustrating call by value for reference types
 */
public class ArgumentTester2
{
    public void doTest()
    {
        BankAccount a = new BankAccount(123, "Fred", 1000);
        addOneDollar(a);
        System.out.println(a);
    }

    private static void addOneDollar(BankAccount b)
    {
        b.deposit(1);
    }
}
4.6 Review of OOP concepts

We can construct a Tester object and invoke its doTest() method. This will call the static addOneDollar method and display the resulting value of a which will show the increase in balance by one dollar.

4.6.9 main method

So far our classes have been designed for execution inside the BlueJ environment but if you try to run one of them outside the BlueJ environment using the command line java interpreter you will get an error message indicating that the interpreter cannot find the infamous main method.

When the interpreter tries to run a class it looks for a special static method, called the main, method, having the exact form

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

If found the interpreter will begin executing the statements in the body of this method. There are two ways to introduce the main method:

- Put a main method in the class you want to run. If your program consists of more than class one of them should have a main method and it will be called the main class.

- Write a special runner class that has a main method and run it with the interpreter. This has the advantage that the class you want to run doesn’t need to be modified.

We illustrate both approaches and then show how to run classes containing a main method using a command prompt (Windows), or terminal window (Unix) and the java compiler (javac) and interpreter (java).

Adding a main method to a class

As an example, we make a new project called main-method containing Point, Circle, and CircleTester from project geometry using “Add class from file”. The project is shown in Figure 4.19.

Now modify this version of CircleTester (page 127) by adding the main method

```java
public static void main(String[] args) {
    CircleTester tester = new CircleTester();
    tester.doTest();
}
```

to obtain the class
package chapter4.main_method; // remove this line if you're not using packages
import chapter4.geometry.Point; // remove this line if you're not using packages
import chapter4.geometry.Circle; // remove this line if you're not using packages

/**
 * A short class to show how to test the Circle and Point classes.
 * This version contains a main method.
 */
public class CircleTester
{
    public CircleTester()
    {
    }

    /**
     * Test the Point and Circle classes.
     */
    public void doTest()
    {
        Point center = new Point(3,4);
        Circle c1 = new Circle();
        Circle c2 = new Circle(center, 5);
        Circle c3 = new Circle(3, 4, 5);
        System.out.println("c1 = " + c1);
        System.out.println("c2 = " + c2);
        System.out.println("c3 = " + c3);
        double radius = c2.getRadius();
        double x = c2.getCenter().getX();
4.6 Review of OOP concepts

double y = c2.getCenter().getY();
System.out.println("Radius = " + radius);
System.out.println("Center x = " + x);
System.out.println("Center y = " + y);
}

public static void main(String[] args)
{
    CircleTester tester = new CircleTester();
tester.doTest();
}

This version of the class can be used both inside and outside BlueJ. The purpose of the main method here is to execute the statements we would do interactively with BlueJ: (1) an object called tester is constructed, (2) the doTest method is invoked on it. The main method can even be executed from within BlueJ. Just right click on the CircleTester rectangle and you will see void main(args) on the class menu. Select it and click OK to get them output as you would be creating an object with the constructor and invoking its doTest method.

Writing a runner class

The second approach which doesn’t involve changing the class you want to run is to write a special runner class that has a main method in it. Example 4.21 shows a simple test of the BankAccount class using BeanShell. We can write this test as the following class:

```java
package chapter4.main_method; // remove this line if you're not using packages
import chapter4.bank_account.BankAccount; // remove this line if you're not using packages
/**
 * A class with only a main method showing how to do a simple test
 * of the BankAccount class outside BlueJ. The class is written so
 * that it can also be tested within BlueJ
 */
public class BankAccountRunner
{
    public void doTest()
    {
        BankAccount account = new BankAccount(123, "Peter Pascoe", 125.50);
        System.out.println("Initial balance is " + account.getBalance());
        account.withdraw(100);
        System.out.println("Balance is " + account.getBalance());
        account.deposit(100);
        System.out.println("Balance is " + account.getBalance());
        System.out.println(account);
    }
}
```
public static void main(String[] args)
{
    BankAccountRunner runner = new BankAccountRunner();
    runner.doTest();
}

We have written this class so that it can also be run inside BlueJ in the usual way: (1) create a
BankAccountRunner object called runner using the new BankAccountRunner() menu choice
from the yellow class rectangle, (2) select the doTest choice from the object menu. The main
method does exactly the same thing.

4.7 Running a class with a main method

To run a class with a main method outside BlueJ it is necessary to open a terminal window or
console window (called an MS-DOS prompt or command prompt in windows) and then compile
and interpret the class using the javac and java commands.

You can now navigate to the directory c:/book-projects/chapter4/main_method and is-
sue the compiler command

    javac BankAccountRunner.java

This command produces the bytecode file (see Chapter 1) called BankAccountRunner.class. You
can use BlueJ do this step too and then open a command prompt and just use the interpreter.

To run the bytecode file use the interpreter command

    java BankAccountRunner

The following output should appear in the terminal window.

    Initial balance is 125.5
    Balance is 25.5
    Balance is 125.5
    BankAccount[number=123, name=Peter Pascoe, balance=125.5]

4.8 Review exercises

- Review Exercise 4.1 Define the following terms and give examples of each.

  class implementation  class specification  public interface
  unit testing           mutable class       immutable class
  Math.random            java.util.Random    Java package
  fully qualified name   this                association
  aggregation            toString            Date
  Calendar               SimpleDateFormat  NumberFormat
  DecimalFormat          constructor call expression  factory method
4.9 **BeanShell exercises**

- **Review Exercise 4.2** How can you easily distinguish a constructor prototype from a method prototype?

- **Review Exercise 4.3** Explain the relationship between a constructor declaration, a constructor prototype, and a constructor call expression.

- **Review Exercise 4.4** Explain the relationship between a method declaration, a method prototype, and a method call expression.

- **Review Exercise 4.5** Give three examples of a constructor prototype and for each example give three examples of a constructor call expression.

- **Review Exercise 4.6** Give three examples of a method prototype and for each example give three examples of a method call expression.

- **Review Exercise 4.7** If you do not provide a constructor in a class the compiler will provide a default constructor. What does this constructor do?

- **Review Exercise 4.8** What is the difference between a default constructor and a no-arg constructor?

- **Review Exercise 4.9** What is the difference between the public interface of a class and its implementation?

### 4.9 BeanShell exercises

- **BeanShell Exercise 4.1** Write some statements that define a string index, extract the character at that index, and display it.

- **BeanShell Exercise 4.2** Write some statements that define a string index, extract the character at that index as a one-character string, and display it.

- **BeanShell Exercise 4.3** Write some statements that define a start index and an end index, extract the substring whose first character is at the start index and whose last character is at the end index, and display the substring.
BeanShell Exercise 4.4 Write some statements that define two strings, search one string for the other string, and display the result.

BeanShell Exercise 4.5 Write some statements that convert a given string s, assumed to contain 4 lowercase letters, to a string called alternate of alternate upper and lower case letters. For example, if s has the value "abcd" then alternate has the value "AbCd".

BeanShell Exercise 4.6 Define an all-uppercase string called nameUpper and write a single statement that converts it to a string called name in which the first letter is upper case and the remaining letters are lower case. For example, "WILLIAM" will be converted to "William".

BeanShell Exercise 4.7 Write statements that construct two BankAccount objects each having an initial balance of $1,000. Use the withdraw and deposit methods to withdraw $50 from the first account and deposit it in the second account. Display the results.

BeanShell Exercise 4.8 Write statements that construct two BankAccount objects each having an initial balance of $1,000. Construct a TransferAgent object to transfer $50 from the first account to the second account. Display the results.

BeanShell Exercise 4.9 Write statements using TriangleCalculatorTester (page 118) to test the TriangleCalculator class.

BeanShell Exercise 4.10 Write some statements that use Point (page 119) to construct two points, calculate the distance between them, and display the results. (NOTE: it is necessary to use import Point; in BeanShell so that we get our Point class instead of java.awt.Point)

BeanShell Exercise 4.11 Write some statements that use Circle (page 122) to construct two circles and display the average of the x and y coordinates of their centers. (see Example 4.25)

BeanShell Exercise 4.12 Write some statements that use the Calendar class to determine what day of the week you were born on.

BeanShell Exercise 4.13 Write some statements that define a double number and use the DecimalFormat class to display this number in fixed format with 3 digits after the decimal point.

BeanShell Exercise 4.14 Write some statements that define a double number and use the DecimalFormat class to display this number in scientific format with 3 digits after the decimal point.

BeanShell Exercise 4.15 The NumberFormat class has a locale dependent static method called getNumberInstance that formats fixed point numbers for your locale. Experiment with the following statements:

```java
NumberFormat fix5 = NumberFormat.getNumberInstance();
fix5.setMinimumFractionDigits(5);
fix5.setMaximumFractionDigits(5);
```

BeanShell Exercise 4.16 Write some statements to show the difference between call by value for primitive types and for a mutable reference type.
4.10 Programming exercises

In each programming exercise you should include javadoc comments and indicate what data you have used to test your class.

► Exercise 4.1 (A FullNameMaker class)
Write a class called FullNameMaker with the following interface.

```java
public class FullNameMaker
{
    // data fields go here
    public FullNameMaker(String first, String mid, String last) {...}
    public String getName1() {...}
    public String getName2() {...}
    public String getName3() {...}
}
```

Here the three names of a person are provided to the constructor. Method getName1 returns the full name, method getName2 returns the full name but with the middle name replaced by its first letter followed by a period, and method getName3 returns the last name first, followed by a comma and a space followed by the first name, followed by a space, the middle initial and a period. For example, if the three input names are "WILLIAM", "James", and "duncan" then the methods should return the strings "William James Duncan", "William J. Duncan", and "Duncan, William J.". All names are stored and output with the first letter capitalized and other letters in lower case regardless of the input.

► Exercise 4.2 (A CopyCard class)
Make a project called copy-card. Using BankAccount as a model write a class called CopyCard that represents a student photocopy card using the student number, studentID (as an integer), the student name, name (as a String), and the amount remaining on the card, balance (as a double number). Include two methods that change the amount on the card: (1) an add method that adds a given amount to the copy card balance and (2) a subtract method that subtracts a given amount. The three “get” methods each return one of the data field values.

Also include in the copy-card project the following tester class that has a doTest method for use inside BlueJ and a main method for use outside BlueJ.

```java
public class CopyCardTester
{
    public void doTest()
    {
        CopyCard fred = new CopyCard(123, "Fred Bolger", 125.0);
        CopyCard mary = new CopyCard(456, "Mary Nelson", 150.0);
        fred.subtract(25.0);
        mary.subtract(50.0);
        displayCard(fred);
        displayCard(mary);
        fred.add(20.0);
    }
}
```
Here we use a private method to display the results after each operation on the card. You can also run this class in BeanShell as follows. Make sure you select “Capture System in/out/err” from the BeanShell File menu and use addClassPath for your project. Then use the statements

```java
CopyCardTester tester = new CopyCardTester();
tester.doTest();
```

▶ Exercise 4.3 (A Triangle class)

Make a project called triangle that contains the Point class from project

```java
import project-book-projects/chapter4/geometry
```

Add to this project the Triangle class with the public interface

```java
public class Triangle
{
    public Point v1, v2, v3;
    public Triangle(Point p1, Point p2, Point p3) {...}
    public Point getVertex1() {...}
    public Point getVertex2() {...}
    public Point getVertex3() {...}
    public double area() {...}
    public String toString() {...}
}
```

that uses Point objects (aggregation) for the three vertices v1, v2, and v3. To calculate the area use the interesting formula

\[
area = \frac{1}{2} \left| (x_2y_3 - x_3y_2) + (x_3y_1 - x_1y_3) + (x_1y_2 - x_2y_1) \right|
\]

where \((x_1,y_1), (x_2,y_2),\) and \((x_3,y_3)\) are the three vertices. Recall that the absolute value of a number \(x\) is denoted by \(|x|\). The method `Math.abs` can be used to calculate it.
Exercise 4.4 (A Name class)
Make a BlueJ project called person. Write a class called Name that represents the name of a person using the following design.

```java
public class Name {
    private String firstName;
    private String lastName;

    public Name(String first, String last) {...}
    public Name(String fullName) {...}

    public String getFirstName() {...}
    public String getLastName() {...}
    public String toString() {...}
}
```

For the second constructor assume that a full name is the first name followed by one space followed by the last name. You can extract the two names using `indexOf` to find the position of the space.

Also, store the first and last names in a standard form. For example, “james”, ”James”, and ”JAMES” should be stored as ”James”.

Exercise 4.5 (An Address class)
Continuing Exercise 4.4, add to the person project an Address class that represents the address of a person using the following design.

```java
public class Address {
    String street;
    String city;
    String province;
    String postalCode;

    public Address(String street, String city, String province, String postalCode) {...}

    public String getStreet() {...}
    public String getCity() {...}
    public String getProvince() {...}
    public String getPostalCode() {...}
    public String toString() {...}
}
```

Exercise 4.6 (A simpler version of Calendar)
Continuing Exercise 4.4 and Exercise 4.5, add to the person project an adapter class that is a simpler version of the complex Calendar class with the following public interface.
import java.util.Calendar;
/**
 * A simplified version of the complicated GregorianCalendar class that is
 * part of the standard Java library and contains about 50 methods. Our
 * class just needs to deal with the year, month, and the day. A class like
 * this is called an adapter class since it adapts a more complex class for
 * our simpler needs.
 */
public class CalendarDate {
    private Calendar calendar;

    /**
     * Construct a calendar date for right now.
     */
    public CalendarDate() {...}

    /**
     * Construct a calendar date for a given year, month, and day.
     * @param year the year
     * @param month the month in the range 1 to 12
     * @param day the day of the month in range 1 to 31
     */
    public CalendarDate(int year, int month, int day) {...}

    /**
     * Return the year for this date.
     * @return the year for this date
     */
    public int getYear() {...}

    /**
     * Return the month for this date.
     * @return the month for this date in range 1 to 12
     */
    public int getMonth() {...}

    /**
     * Return the day of the month for this date.
     * @return the day of the month for this date in range 1 to 31
     */
    public int getDayOfMonth() {...}

    /**
     * Return a string representation of a calendar date.
     * @return a string representation of a calendar date
     */
4.10 Programming exercises

public class Person
{
    private Name name;
    private Address address;
    private CalendarDate birthDate;

    public Person(Name n, Address a) {...}

    public String getName() {...}
    public String getAddress() {...}
    public CalendarDate getBirthDate() {...}
    public int ageThisYear() {...}
    public String toString() {...}
}

Exercise 4.7 (A Person class using aggregation)
Continuing Exercise 4.4, Exercise 4.5 and Exercise 4.6, add to the person project a Person class
that represents the address of a person using the following design.

Exercise 4.8 (New methods for the Point class)
Make a project called point that contains the Point class from project
book-projects/chapter4/geometry
Include the following methods to the Point class.

(a) an instance method called add that takes a Point object as an argument and adds it to “this”
    Point object, returning a new Point object. Use the addition formula \((a,b) + (c,d) = (a+c, b+d)\).

(b) similarly include a sub method using the subtraction formula \((a,b) - (c,d) = (a-c, b-d)\).

(c) a static method called add that takes two Point objects as arguments, adds them, and returns
    a new Point object.

(d) similarly include a static sub method.