**Using Objects and Classes**

This chapter reinforces the topics introduced in the last two chapters, and discusses several new topics. The Point class class is used as an example.

**Chapter Topics:**

* The Point class
* Class descriptions
* The toString() method
* Changing the data in an object vs constructing a new object
* The equals() method
* The == operator (again)
* Aliases

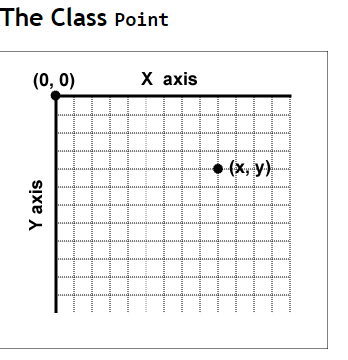
You have seen some of the material in this chapter before. But, don't rush through it. The purpose of this chapter is to review topics that you may still be somewhat unclear about.

**QUESTION 1:**

In geometry, what is a two dimensional point?

### Answer:

A 2D geometrical point gives a location using two values, usually an x value and a y value.



Think of a two dimensional point as an object that consists of two values, the distance x from the origin along a horizontal axis, and the distance y from the origin along a vertical axis. In Java graphics, the upper left corner of a drawing area (such as a window) is considered to be the origin (0, 0), so y increases going downward.

It is convenient to think of a point as a single thing (not as two separate things). For example, you think of the end point of a line as one thing. And you think of the corner of a square as one thing. This is an example of abstraction. In Java, an abstraction usually calls for a class.

Java comes with a package of predefined classes that are used for building graphical user interfaces, the Application Windowing Toolkit, or the AWT. One of the many classes in the AWT is the class Point.

The class Point describes two things: the variables (the data) and the methods (the behavior) of Point objects.

### QUESTION 2:

What two variables do you expect a Point object to have?

### Answer:

A point consists of a pair of numbers, (x, y).

# Description of a Class

A description of the class Point is found in the Java documentation. Look at the documentation on your hard disk or on the web to see it. You will see something like the following:

java.awt

Class Point

. . .

// Field Summary

int x;

int y;

// Constructor Summary

Point(); // creates a point at (0,0)

Point(int x, int y); // creates a point at (x,y)

Point( Point pt ); // creates a point at the location given in pt

// Method Summary

boolean equals(Object obj); // checks if two point objects hold equivalent data

void move(int x, int y); // changes the (x,y) data of a point object

String toString(); // returns character data that can be printed

(I've left out some methods we won't be using.)

The documentation shows the data and methods contained in Point objects, the constructors that create such objects. Variables are sometimes called **fields** (as is done here). The two variables are named x and y and are of type int.

### QUESTION 3:

There are three constructors listed for Point. Each one creates the same type of object. What is the difference between the constructors?

### Answer:

The different constructors require different parameters. (Different types of data are supplied to the constructors.)

# Multiple Constructors

Any of the three constructors can be used to create a Point. It is a matter of convenience which constructor you use. Here is an example:

import java.awt.\*; // import the package that contains Point

class PointEg1

{

public static void main ( String arg[] )

{

Point a, b, c; // reference variables

a = new Point(); // create a Point at (0, 0);

// save the reference in "a"

b = new Point( 12, 45 ); // create a Point at (12, 45);

// save the reference in "b"

c = new Point( b ); // create a Point containing data equivalent

// to the data referenced by "b"

}

}

To use the definition of class Point, import the package that contains it. The statement:

import java.awt.\*

says to use the AWT package that comes with Java. The \* says that everything defined in the package can be used (although this program only uses the Point class).

### QUESTION 4:

Say that the program has just been loaded into main memory and is just about to start running.

How many reference variables are there?

How many objects are there?

**Answer:**

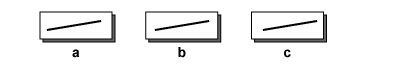
Say that the program has just been loaded and is just about to start running.

How many reference variables are there?    3

How many objects are there?    Zero

**Instantiating the Point Objects**

Here is a picture of the variables just as the program starts running. No objects have been instantiated yet, so the reference variables a, b, and c do not refer to any objects. To emphasize this, a slash has been put through the box for each variable.



Here is the program again:

import java.awt.\*;

class PointEg1

{

public static void main ( String arg[] )

{

Point a, b, c; // reference variables

a = new Point(); // create a Point at (0, 0);

// save the reference in "a"

b = new Point( 12, 45 ); // create a Point at (12, 45);

// save the reference in "b"

c = new Point( b ); // create a Point containing data equivalent

// to the data referenced by "b"

}

}

The program:

1. Declares three reference variables a, b, and c, which may refer to objects of type Point.
2. Instantiates a Point object with x=0 and y=0.  
   (The documentation tells us that a constructor without parameters initializes x and y to zero.)
3. Saves the reference to the object inthe variable a.
4. Instantiates a Point object with x=12 and y=45.
5. Saves the reference to the object in the variable b.
6. Instantiates a third Point object.
7. Saves the reference in the variable c.

Once each Point object is instantiated, it is the same as any other (except for the values in the data). It does not matter which constructor was used to instantiate it.

**QUESTION 5:**

What are the values of x and y in the third point?

### Answer:

x = 12 and y = 45.

# Picture of the Objects

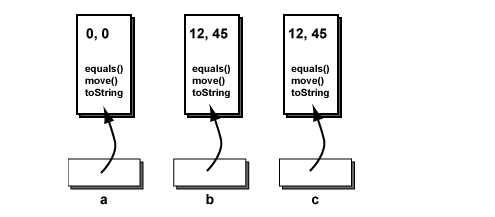
Points b and c were constructed with the statements

b = new Point( 12, 45 );

c = new Point( b );

The constructor for point c is supplied with data in the form of point b. It seems reasonable that the x and y of c will take their values from b. (To be sure, check the[documentation.](http://www.webmastercert.com/moodle/mod/book/ch27_3.html#Point,%20class) )

After the three objects have been created (just before the program closes) the situation looks like this:



Each reference variable refers to an object. Each object contains its own data and its methods. (For clarity, only some methods are shown.) An object does not contain any constructors.

### QUESTION 6:

Look at the picture. Is it clear what the phrase "the object a" means?

### Answer:

A parameter is an item of data supplied to a method or a constructor.

# Program that uses the toString() Method

The documentation shows that toString() needs no parameters. However, you must use parentheses when the method is called. Here is the example program:

import java.awt.\*;

class PointEg2

{

public static void main ( String arg[] )

{

Point a, b, c; // reference variables

a = new Point(); // create a Point at ( 0, 0);

// save the reference in "a"

b = new Point( 12, 45 ); // create a Point at (12, 45);

// save the reference in "b"

c = new Point( b ); // create a Point

String strA = a.toString(); // create a String object based on the data

// found in the object referenced by "a".

System.out.println( strA );

}

}

When this program runs, the statement:

String strA = a.toString(); // create a String object based on the data

creates a String object based on the data in the object referred to by a. The strA refers to this new String object. Then the characters from the String are sent to the monitor with println. The program prints out:

java.awt.Point[x=0,y=0]

The Point object has not been altered: it still exists and is referred to by a. Sometimes people say that toString() converts the Point object to a String, but this is sloppy. Nothing has been converted, but a new object has been created.

### QUESTION 8:

Just as the program is about to end, how many objects have been created? Has any garbage been created?

**Answer:**

Just as the program is about to close, how many objects have been created?     Four, counting the String object.

Has any garbage been created?    No, because a reference variable points to each object.

**Using a Temporary Object**

In the example program, a String object was created, and a reference to it was kept in a reference variable. Here is another modification to the example program:

import java.awt.\*;

class PointEg3

{

public static void main ( String arg[] )

{

Point a = new Point(); // declarations and construction combined

Point b = new Point( 12, 45 );

Point c = new Point( b );

System.out.println( a.toString() ); // create a temporary String based on "a"

}

}

This program creates three Points with the same values as before, but now the declaration and construction of each point is combined.

The last statement has the same effect as the last two statements of the previous program:

1. When the statement executes, a refers to an object with data (0,0).
2. The toString() method of that object is called.
3. The toString() method creates a String object and returns a reference to it.
4. At this point of the execution, you can think of the statement like this:

System.out.println( reference to a String );

1. The println method of System.out uses the reference to find the data to print out on the monitor.
2. The statement finishes execution; the reference to the String has not been saved anywhere.

Since the String reference was not saved in a reference variable, there is now no way to find it. It is garbage. That is OK. It was only needed for one purpose, and that purpose is completed. Using objects in this manner is very common.

**QUESTION 9:**

What type of parameter (stuff inside parentheses) does the System.out.println() method expect?

### Answer:

Yes. The method just returns a reference to the String itself. No new object is created.

# Another Example Program

Here is the example program with yet another change:

import java.awt.\*;

class PointEg3

{

public static void main ( String arg[] )

{

Point a = new Point(); // declarations and construction combined

Point b = new Point( 12, 45 );

Point c = new Point( b );

System.out.println( a ); // create a temporary String based on "a", print it out

System.out.println( b ); // create a temporary String based on "b", print it out

System.out.println( c ); // create a temporary String based on "c", print it out

}

}

The program prints out:

java.awt.Point[x=0,y=0]

java.awt.Point[x=12,y=45]

java.awt.Point[x=12,y=45]

This program is deceptively short; its execution calls for quite a bit of activity.

### QUESTION 11:

Just as the program is about to close, how many objects have been created?

How many object references are there?

Has any garbage been created?

### Answer:

Just as the program is about to close, how many objects have been created ?     Six — three Point objects and three temporary String objects

How many object references are there?     Three — each referencing a Point

Has any garbage been created?    Three objects — each temporary String object

# Changing Data inside a Point

Look again at the description of class Point. One of the methods is:

public void move( int x, int y ) ;

This method is used to change the x and the y data inside a Point object. The modifier public means that it can be used anywhere in your program; void means that it does not return a value. This part of the description

( int x, int y )

says that when you use move, you need to supply two int parameters that give the new location of the point. A parameter is information you supply to a method.

Here is the example program, modified again:

import java.awt.\*;

class PointEg4

{

public static void main ( String arg[] )

{

Point pt = new Point( 12, 45 ); // construct a Point

System.out.println( pt );

pt.move( -13, 49 ) ; // change the x and y in the Point

System.out.println( pt );

}

}

Here is what it writes to the screen:

java.awt.Point[x=12,y=45]

java.awt.Point[x=-13,y=49]

### QUESTION 12:

How many Point objects are created by this program?

How many temporary String objects are created by this program?

### Answer:

How many Point objects are created by this program?     One — the object referenced by pt

How many temporary String objects are created by this program?     Two — one temporary String object for each println()

# One Object, with Changing Data

The program changes the data inside a Point object using that object's move() method:

import java.awt.\*;

class PointEg4

{

public static void main ( String arg[] )

{

Point pt = new Point( 12, 45 ); // construct a Point

System.out.println( pt );

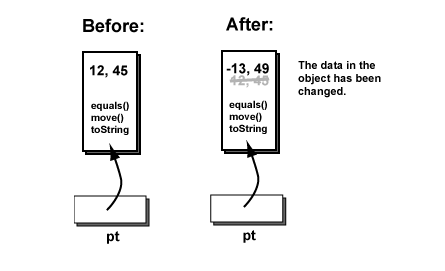
pt.move( -13, 49 ) ; // change the x and y in the Point

System.out.println( pt );

}

}

Here is a picture of this:



### QUESTION 13:

Can a constructor be used to change the data inside an object?

### Answer:

No. Constructors always create new objects. (They might get data from an old object, but a completely separate object will be constructed using a different chunk of main memory.)

# Dangerously Similar Program

Here is the example program, this time modified to create a second Point object:

import java.awt.\*;

class ChangingData2

{

public static void main ( String arg[] )

{

Point pt = new Point( 12, 45 ); // construct a Point

System.out.println( pt );

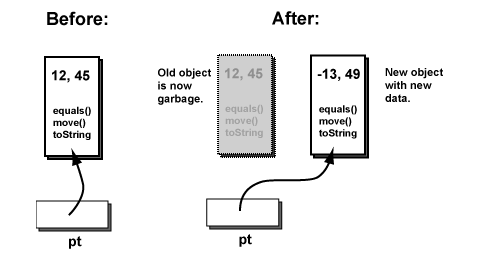
pt = new Point( -13, 49 ) ; // construct a new Point

System.out.println( pt );

}

}

Here is a picture showing the situation before and after the second assignment statement:



In the "After" picture, the first object is shaded to emphasis that it is now "garbage." The reference variable pt refers to the newly created object.

### QUESTION 14:

What will this second version of the program output to the monitor?

### Answer:

Exactly the same as the first version of it:

java.awt.Point[x=12,y=45]

java.awt.Point[x=-13,y=49]

You can not be sure about the internal workings of a program by inspecting its output!

# Review

You might want to review the two programs to be sure that you understand their differences:

|  |  |
| --- | --- |
| **Program:** | **What it did:** |
| [The first program](http://www.webmastercert.com/moodle/mod/book/view.php?id=5205&chapterid=1161#object,%20method%20changes%20state) | An object was constructed. The variable pt refers it. Then, new data replaced the old data inside the same object. |
| [The second program](http://www.webmastercert.com/moodle/mod/book/view.php?id=5205&chapterid=1162#object,%20assignment%20replaces%20old) | An object was constructed. The variable pt refers it. Then, a new object was constructed with new data, and pt was changed to refer to the new object. The first object became garbage. |

### QUESTION 15:

When are two points equivalent?

### Answer:

The x value and the y value of each point is the same.

# The equals() Method

The equals() method is defined for class Point to perform this test:

pointA.equals( pointB ) —— returns true if the two points

contain equivalent data

The example program shows this:

import java.awt.\*;

class EqualsDemo

{

public static void main ( String arg[] )

{

Point pointA = new Point( 7, 99 ); // first Point

Point pointB = new Point( 7, 99 ); // second Point with equivalent data

if ( pointA.equals( pointB ) )

System.out.println( "The two objects contain the same data: " + pointA );

else

System.out.println( "The two objects are not equivalent: " + pointA +

" differs from" + pointB);

}

}

### QUESTION 16:

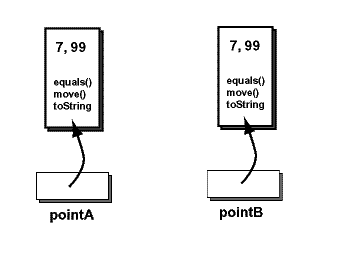
What is the output of this program? (You might wish to copy-paste-and-run this program to check your answer.)

### Answer:

The two objects contain the same data: java.awt.Point[x=7,y=99]

# Different Objects, Equivalent Data

Here is a picture showing the situation. There are two distinct objects (each constructed out of different bytes of memory). The equals() method returns true because the data is equivalent.



### QUESTION 17:

Could the == operator be used in this program instead of the equals() method?

### Answer:

No. The == operator tests if two reference variables refer to the same object.

# Testing Two Reference Variables

Here is the program, with changes:

import java.awt.\*;

class EqualsDemo2

{

public static void main ( String arg[] )

{

Point pointA = new Point( 7, 99 ); // first Point

Point pointB = new Point( 7, 99 ); // second Point with equivalent data

if ( pointA == pointB )

System.out.println( "The two variables refer to the same object" );

else

System.out.println( "The two variables refer to different objects" );

}

}

The picture of the situation (after the two new operators have executed) is the same as on the previous page.

### QUESTION 18:

What is the output of this program?

### Answer:

The two variables refer to different objects

# Alias

It is possible to have two (or more) reference variables refer to the same object. Here is a modification to the program that does that:

import java.awt.\*;

class EqualsDemo3

{

public static void main ( String arg[] )

{

Point pointA = new Point( 7, 99 ); // pointA refers to a Point Object

Point pointB = pointA; // pointB refers to the same Object

if ( pointA == pointB )

System.out.println( "The two variables refer to the same object" );

else

System.out.println( "The two variables refer to different objects" );

}

}

Only one object has been created (because there is only one new operator.) The second statement:

Point pointB = pointA;

copies the reference that is in pointA into the reference variable pointB. Now both reference variables lead to the same object.

**alias:** When two or more reference variables refer to the same object, each variable is said to be an alias.

### QUESTION 19:

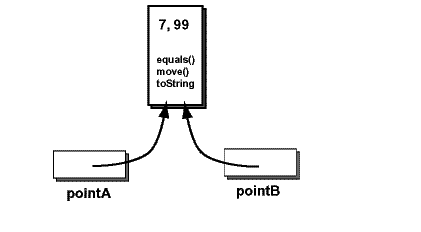
What is the output of the program?

### Answer:

The two variables refer to the same object

# alias-detector

Remember that the == operator looks only at variables. If two variables contain a reference to the same object, the operator evaluates to true. **Think of the == operator as an alias-detector**. Here is a picture of the situation of the last example program:



### QUESTION 20:

(Thought questionsmile Could the equals() method be used with aliases, as in this program?

**Answer:**

Yes. You have to be careful how you use the result, however.

**Equivalence of Aliases**

Here is the example program (again!) with this modification:

import java.awt.\*;

class EqualsDemo4

{

public static void main ( String arg[] )

{

Point pointA = new Point( 7, 99 ); // pointA refers to a Point Object

Point pointB = pointA; // pointB refers to the *same* Object

if ( pointA.equals( pointB ) )

{

System.out.println( "The two variables refer to the same object," );

System.out.println( "or different objects with equivalent data." );

}

else

System.out.println( "The two variables refer to different objects" );

}

}

The picture of the situation is the same as before. But now the equals() method is used. It:

1. Uses the reference pointA to get the x and y from an object.
2. Uses the reference pointB to get the x and y from an object.
3. Determines that the two x's are the same and that the two y's are the same.
4. Returns a *true*.

The fact that the object is the same object in step 1 and step 2 does not matter. The x's that are tested are the same, and the y's that are tested are the same, so the result is *true*.

**QUESTION 21:**

If the == operator returns a *true* will the equals() method return a *true*, always?

### Answer:

Yes.

# Summary

The following table is a summary. Assume that each row is independent of the others.

|  |  |  |
| --- | --- | --- |
| **code section** | **pointA == pointB** | **pointA.equals( pointB )** |
| Point pointA = new Point( 21, 17 );  Point pointB = new Point( 21, 17 ); | false | true |
| Point pointA = new Point( 21, 17 );  Point pointB = new Point( -99, 86 ); | false | false |
| Point pointA = new Point( 21, 17 );  Point pointB = pointA; | true | true |

### QUESTION 22:

Will pointA.equals(pointB) return the same true/false value as pointB.equals(pointA) ?

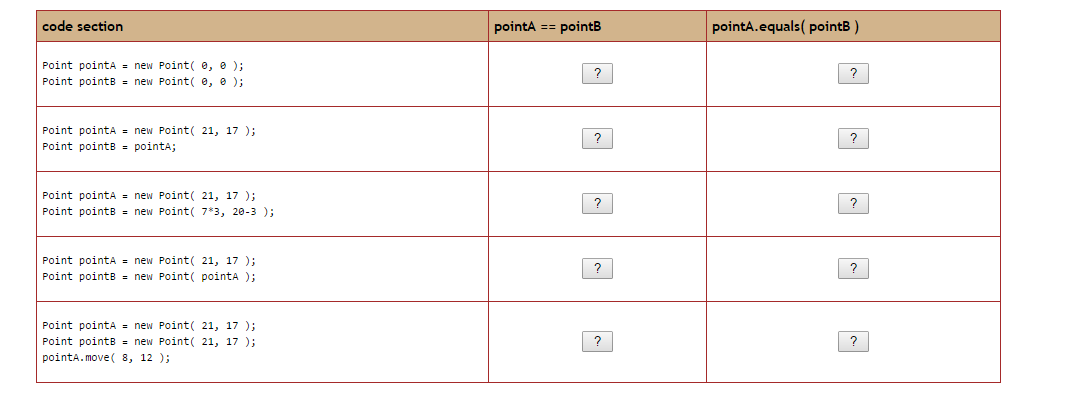
### Answer:

Yes.

# Practice

Perhaps you would like to practice? Assume that each row is independent of the others.

Top of Form



Bottom of Form

### QUESTION 23:

Do you imagine that professional programmers ever get == and equals confused?

**Parameters in Method Calls**

Often when you use a method, you supply it with some information in the form of parameters. This chapter discusses method calls and parameters.

**Chapter Topics:**

* Methods with no parameters
* Parameter lists
* Data types of parameters
* Expressions in parameter lists
* Conversion of values in parameter lists to required types

**QUESTION 1:**

After the following statements have executed,

String str = new String("alphabet soup");

int len = str.length();

what value is in the variable len?

**Answer:**

13

**Expressions in Parameter Lists**

In this case, saying what method to run is not enough. The method needs information about what it is to do. When you look at the description of class Point the move() method is described as follows:

public void move(int x, int y); // change (x,y) of a point object

This description says that the method requires two parameters:

1. The first parameter is type int. It will become the new value for x.
2. The second parameter is type int. It will become the new value for y.

Dot notation is used to specify which object, and what method of that object to run. The parameter list of the method supplies the method with data. Here are some examples:

Point pointA = new Point(); Point pointB = new Point( 94, 172 ); pointA.move( 45, 82 ); int col = 87; int row = 55; pointA.move( col, row );

You can put *expressions* into parameter lists as long as the expression evaluates to the type expected by the method. Of course, the expressions are evaluated (using the usual rules) before the method starts running.

pointB.move( 24-12, 30\*3 + 5 ); pointB.move( col-4, row\*2 + 34 );

**QUESTION 3:**

What is the location of pointB after the following:

pointB.move( 24-12, 30\*3 + 5 );

### Answer:

pointB will now be located at x = 12, y = 95.

# Step-by-Step Method Call

You probably did the right thing to get the answer, but let us go through it again, just to be sure:

pointB.move( 24-12, 30\*3 + 5 );

is equivalent to:

pointB.move( 12, 30\*3 + 5 );

is equivalent to:

pointB.move( 12, 90 + 5 );

is equivalent to:

pointB.move( 12, 95 );

At this point, the move() method starts running with the two int values it requires.

The expressions in the parameter list are evaluated before the method starts running. The resulting values should be the data type expected by the method, or a data type that can be converted to that type.

### QUESTION 4:

What do you suspect will happen with the following method call?

pointB.move( 14.305, 34.9 );

**Answer:**

This is an error, because the parameter list has two floating point values, not the two int values required.

**Parameters Must be the Correct Type**

When a method starts running, it must have the right number of parameters, and each parameter must be of the required type. However, sometimes, just before a method starts running, the values supplied in a parameter list are *converted* to the required type. There are two ways in which this can happen:

1. Explicitly — the programmer asks for values to be converted with a *type cast*.
2. Implicitly — if the compiler can make the conversion without loss of information, or with only a small loss of precision, it will automatically do so.

A **type cast** looks like this:

(requiredType)(expression)

The (requiredType) is something like (int). The expression is an ordinary expression. If it is a single variable, you don't need the surrounding parentheses. Here is an example program that shows a type cast:

import java.awt.\*; // import the package where Point is defined

class TypeCastEg

{

public static void main ( String arg[] )

{

Point pointB = new Point(); // create a point at x=0 y=0

pointB.move( (int)14.305, (int)(34.9-12.6) );

System.out.println("New location:" + pointB );

}

}

In this case, a type cast is required for both parameters because converting a floating point number to an int will usually lose information. In *casting* a floating point value to an int, the fractional part will be lost. (*Not* rounded.)

**QUESTION 5:**

What will be the output of the program?

### Answer:

New location:java.awt.Point[x=14,y=22]

# Automatic Conversion of Parameter Type

In the previous example, converting from floating point to int results in a loss of information, so the programmer must explicitly ask for conversion with a type cast.

When a conversion from one type to another type can be done without loss of information, the compiler will do it automatically. For example, the description of the move()method says that it requires two int parameters:

public void move(int x, int y); // change (x,y) of a point object

An int value is held in 32 bits. A short value that is held in 16 bits can be converted to 32 bits without loss of information.

Why? The information encoded in 16 bits can also be encoded in 32 bits. So the following program will compile and run correctly:

import java.awt.\*; // import the class library where Point is defined

class AutoConvEg1

{

public static void main ( String arg[] )

{

Point pointB = new Point(); // create a point at x=0 y=0

short a=12, b=42;

pointB.move( a, b ); // values in parameter list automatically

// converted to the required type, int.

System.out.println("New location:" + pointB );

}

}

The values inside variables a and b are not changed. Those values are retrieved, then converted to the types that the parameters require.

### QUESTION 6:

What will this program write to the monitor?

**Answer:**

New location:java.awt.Point[x=12,y=42]

**Safe Conversions**

In general, if information might be LOST, a conversion from one type to another will NOT be performed automatically. A conversion from a data type that uses N bits to a type that uses fewer than N bits risks information loss, and will NOT be performed automatically. The compiler makes this decision by examining the data types involved, not the actual values involved.

In the following situations, the compiler will automatically convert from the type of the expression in a method call to the required type:

|  |
| --- |
| **Automatic Conversions** |
| * Converting an integer type to another integer type that uses more bits. * Converting a floating point type to another floating point type that uses more bits. * Converting an integer type to a floating point type that uses the same number of bits  may result in a loss of precision, but will be done automatically. * Converting an integer type to a floating point type that uses more bits will not result in loss of precision  and will be done automatically. |

"Loss of precision" means that some of the less significant digits may become zeros, but the most important digits and the size of the number will remain. Recall (from chapter 8) that float has only about seven decimal digits of precision. For example, converting the int 123456789 to the float 123456700.0 shows a loss of precision (but is done, anyway).

**QUESTION 7:**

Will there be a loss of precision in converting the int 123456789 to a double?

**Answer:**

No, because double uses 64 bits, the equivalent of about 15 decimal digits of precision.

**Unsafe Conversions**

In the following situations, the compiler *will not* automatically perform the conversion.

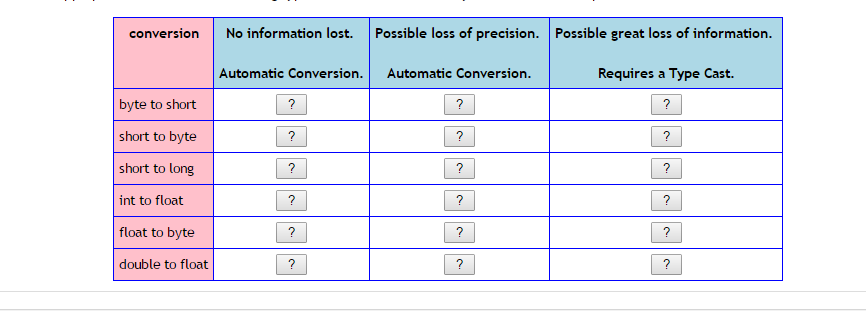
|  |
| --- |
| **NO Automatic Conversion** |
| * Converting from an integer type to another integer type with fewer bits. * Converting from a floating point type to another floating point type with fewer bits. * Converting from a floating point type to any integer type will possibly lose both precision and magnitude. * Conversion to or from boolean is never allowed (not even with a type cast.) |

Remember the sizes of the various primitive types (the primitive types char and boolean are not listed here):

|  |  |
| --- | --- |
| **type** | **number of bits** |
| byte | 8 |
| short | 16 |
| int | 32 |
| long | 64 |
| float | 32 |
| double | 64 |

**QUESTION 8:**

Click the button in the appropriate column for the following type conversions. There is only one correct column per row.



### Answer:

### 

# Example Method Calls

Where there is no loss of information, or only a possible loss of precision, Java will automatically convert the value in a method call to what is required. When there is a possible loss of both precision and magnitude, Java requires you to explicitly state that you want to go ahead with the conversion by using a type cast.

For example:

Point A = new Point();

short a = 93, b = 92;

int i = 12, j = 13;

long x = 0, y = 44;

double u = 13.45, v = 99.2;

A.move( i, j ); // OK --- no conversion needed.

A.move( a, b ); // OK --- shorts converted to ints with no loss

A.move( a, j ); // OK --- short converted to int with no loss

A.move( x, y ); // NOT OK --- possible loss of

// precision and magnitude.

A.move( (int)x, (int)y ); // OK --- type casts say to go ahead

A.move( i, v ); // NOT OK -- possible loss of

// precision and magnitude (for v).

A.move( i, (int)v ); // OK --- type cast says to go ahead.

In making these decisions, it is only the type of the variable (or the expression) that is examined, NOT the particular value. For example, in the first method call marked NOT OK, the particular values could (in this case) be converted from long to int without loss of information. But for other values there would be total loss of information, so the call requires a type cast.

Conversion is done on individual parameters, if needed. In the third call, only the first parameter needed to be converted.

### QUESTION 9:

Which of the following method calls are correct?

Point A = new Point();

short a = 93, b = 92;

int i = 12, j = 13;

long x = 0, y = 44;

double u = 13.45, v = 99.2;

A.move( i, b ); //

A.move( a, x ); //

A.move( u, b ); //

### Answer:

Point A = new Point();

short a = 93, b = 92;

int i = 12, j = 13;

long x = 0, y = 44;

double u = 13.45, v = 99.2;

A.move( i, b ); // OK --- b can be converted to int without loss

A.move( a, x ); // NOT OK --- converting x to int may lose magnitude and precision

A.move( u, b ); // NOT OK --- converting u to int may lose magnitude and precision

# Method using a double Parameter

The cos() method is a static method of the Math class. (Remember that a static method is a method that belongs to the definition of the class; you do not need an object to use it.) Here is how the method looks in the documentation for the Math class:

public static double cos( double num )

The method requires a parameter of type double. It evaluates to (returns) a double, the cosine of the parameter. The parameter is in radians (not degrees.) Here is it being used in a program:

class CosEg

{

public static void main ( String arg[] )

{

double x = 0.0;

System.out.println( "cos is:" + Math.cos( x ) );

}

}

### QUESTION 10:

Is the parameter in the method call the correct type?

**CHAPTER 29B — More about Strings**

String objects are frequently used in programs. This chapter provides extra practice in using them.

**Chapter Topics:**

* Strings are Immutable
* Indexing Strings
* substring()
* indexOf()

Note: the String class is one of the classes that Advanced Placement Computer Science students are expected to know well. This chapter was written after the author observed an unfortunate number of mistakes involving Strings in the APCS 2008 examination.

**QUESTION 1:**

What does the following code write?

class ImmDemo

{

public static void main ( String[] args )

{

String str = new String("I recognize the vestiges of an old flame.");

str.substring( 16 );

System.out.println( str );

}

}

### Answer:

The program writes out:

I recognize the vestiges of an old flame.

The code is syntactically correct and will compile and run, but what it does might not be what the author intended. (The author probably intended to write out a substring of the phrase.)

# Strings are Immutable!

Programmers often forget that String objects are immutable. Once a String object has been constructed, it cannot be changed. This line of code:

str.substring( 16 );

creates a new object, containing a substring of the characters of the original object. However, the original object is not changed. Since the reference variable str continues to point to the original object, the new object immediately becomes garbage. The next statement

System.out.println( str );

writes out the characters in the original object.

### QUESTION 2:

How would you modify the program so that the new substring is written?

**Answer:**

class ImmDemo

{

public static void main ( String[] args )

{

String str = new String("I recognize the vestiges of an old flame.");

str = str.substring( 16 );

System.out.println( str );

}

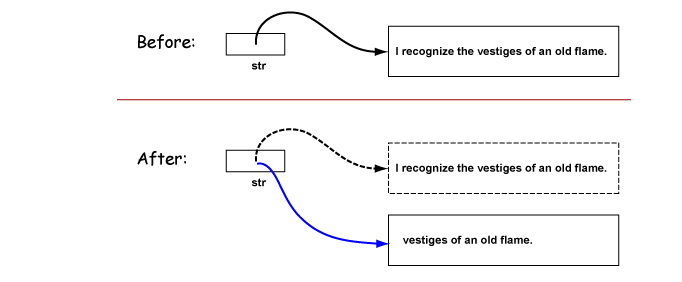
}

**"Changing" a String**

A common mistake is to think "change a string", but to then attempt to change an immutable object. When you think "change a string" you need to do two things:

1. Compute a new String object.
2. Assign the reference to the new String to a reference variable.

The diagram shows how the second version of the program works. The reference variable str is first assigned a reference to the original object. Then a new object is constructed by substring(). The new reference is assigned to str. Then the println() method is called with the new reference, so the new string is written.



**QUESTION 3:**

Which character corresponds to index 0 in the following string?

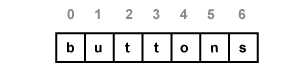
I recognize the vestiges of an old flame.

### Answer:

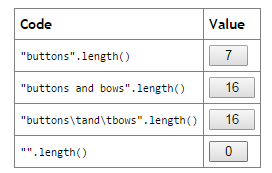
Character 'I' (the beginning character of the string).

# String Indexing

The beginning character of a string corresponds to index 0 and the last character corresponds to the index (length of string)-1.



The length of a string is the number of characters it contains, including spaces, punctuation, and control characters. Fill in the following table by clicking on the buttons.



Top of Form

Bottom of Form

The sequence "\t" stands for a single character, a tab character. Each tab character counts as one character (although it might be displayed on a monitor or printer using several spaces).

Be careful about the difference between an empty string, and a null reference. An empty string is an object that contains no characters, and so has length 0. A null reference means there is no object present.

### QUESTION 4:

What does the following code write?

String myString = null;

System.out.println("The length is: " + myString.length() )

**Answer:**

The program will throw a NullPointerException (and usually stop running).

**Versions of substring()**

There are two versions of substring:

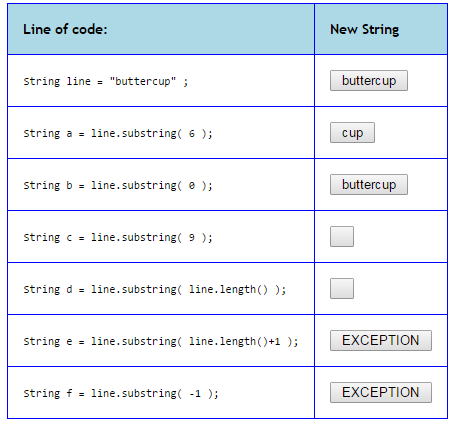
|  |  |  |
| --- | --- | --- |
| substring( int from ) | Create a new object that contains the characters of the method's string from index from to the end of the string. | Throws an IndexOutOfBoundsException if from is negative or larger than the length of the string. |
| substring( int from, int to ) | Create a new object that contains the characters of the method's string from index from to index to-1. | Throws an IndexOutOfBoundsException if from is negative or if from is larger than to. |

There are some tricky rules about the first version of the method:

1. If from is exactly equal to the length of the original string, a substring is created, but it contains no characters (it is an empty string).
2. If from is greater than the length of the original string, or a negative value, a IndexOutOfBoundsException is thrown (and for now, your program crashes).

**QUESTION 5:**

What do the following statements create?



**Instructions:** For each question, choose the single best answer. Make your choice by clicking on its button. You can change your answers at any time. When the quiz is graded, the correct answers will appear in the box after each question.

Top of Form

1. What will the following write?

String str = "Hello World!" ;

str.substring(6);

System.out.println( str );

 **A.** World!

 **B.** Hello

 **C.** Hello World!

 **D.** The fragment will not compile.



Bottom of Form

Top of Form

2. Examine this code:

String str = "Hello World!" ;

System.out.println( str.substring(6) );

What will it write?

 **A.** World!

 **B.** Hello

 **C.** Hello World!

 **D.** The fragment will not compile.



Bottom of Form

Top of Form

3. Examine this code:

String str = "Hello World!" ;

System.out.println( str.length() );

What will it write?

 **A.** 0

 **B.** 10

 **C.** 11

 **D.** 12



Bottom of Form

Top of Form

4. Examine this code:

String myString = "";

System.out.println( myString.length() );

What what will it write?

 **A.** 0

 **B.** 1

 **C.** 2

 **D.** the code will not compile



Bottom of Form

Top of Form

5. Examine this code:

String str = "Hello\tWorld!" ;

System.out.println( str.length() );

What will it write?

 **A.** 0

 **B.** 10

 **C.** 11

 **D.** 12



Bottom of Form

Top of Form

6. Examine this code:

String str = "Hello World!" ;

What is the index of the character 'W' ?

 **A.** 0

 **B.** 5

 **C.** 6

 **D.** 7



Bottom of Form

Top of Form

7. Examine this code:

String str = "One Fine Day" ;

String val = str.substring(4) ;

System.out.println( val );

What does the fragment print?

 **A.** One Fine Day

 **B.** Fine Day

 **C.** Day

 **D.** One Fine



Bottom of Form

Top of Form

8. Examine this code:

String str = "One Fine Day" ;

String val = str.substring(0, 3) ;

System.out.println( val );

What does the fragment print?

 **A.** One Fine Day

 **B.** Fine Day

 **C.** One

 **D.** One Fine



Bottom of Form

Top of Form

9. Examine this code:

String str = "One Fine Day" ;

String val = str.substring(0, str.length() ) ;

System.out.println( val );

What does the fragment print?

 **A.** One Fine Day

 **B.** Fine Day

 **C.** One

 **D.** The code with throw an IndexOutOfBoundsException



Bottom of Form

Top of Form

10. Examine this code:

String str = "One Fine Day" ;

String val = str.substring(4, 7 ) ;

System.out.println( val );

What does the fragment print?

 **A.** nothing is printed since val contains an empty string.

 **B.** Fine

 **C.** Fin

 **D.** The code with throw an IndexOutOfBoundsException



Bottom of Form

Top of Form

11. Examine this code:

String str = "One Fine Day" ;

String val = str.substring( 7, 4 ) ;

System.out.println( val );

What does the fragment print?

 **A.** nothing is printed since val contains an empty string.

 **B.** Fine

 **C.** Fin

 **D.** The code with throw an IndexOutOfBoundsException



Bottom of Form

Top of Form

12. Examine this code:

String str = "One Fine Day" ;

String A = str.substring(0,3);

String B = str.substring(3);

System.out.println( A+B );

What does the fragment print?

 **A.** One Day

 **B.** One Fine Day

 **C.** OneFine Day

 **D.** On Fine Day



Bottom of Form

Top of Form

13. Examine this code:

String str = "One Fine Day" ;

String A = str.substring(4);

String B = A.substring(0,4);

System.out.println( B );

What does the fragment print?

 **A.** One

 **B.** Fine

 **C.** Day

 **D.** Fine Day



Bottom of Form

Top of Form

14. Examine this code:

String str = "One Fine Day" ;

System.out.println( str.indexOf( "Fine" );

What does the fragment print?

 **A.** 0

 **B.** 3

 **C.** 4

 **D.** 5



Bottom of Form

Top of Form

15. Examine this code:

String str = "One Fine Day" ;

System.out.println( str.indexOf( "" ) );

What does the fragment print?

 **A.** 0

 **B.** 3

 **C.** 4

 **D.** The code with throw an IndexOutOfBoundsException



Bottom of Form

Top of Form

16. Examine this code:

String str = "One Fine Day" ;

System.out.println( str.substring( str.indexOf("Fine") ) );

What does the fragment print?

 **A.** Fine Day

 **B.** Day

 **C.** One Fine Day

 **D.** The code with throw an IndexOutOfBoundsException



# Chapter 29B Programming Exercises

## Exercise 1 — Pascal to Java Translator

In the Pascal programming language, the assignment operator is the two character sequence := instead of a single = character as in Java. Write a program that reads in a source file line by line. Each sequence := in a line is changed into = before the line is output. (Of course, much more than this is needed to fully translate Pascal to Java.) Use redirection for file input and output:

C:\>java Transformer < Hello.pas > Hello.java

Look at the chapter "File Input and Output" if it is not clear what the command line is doing. Here is a sample Pascal program (copy and paste it to a file for use with your program):

program noparmTest (input, output);

var a, b : integer;

c : real;

function two: result integer;

var n : integer;

begin

c := a + b;

two := -c

end

begin

a := 1;

b := 2;

c := a+b+two;

write(c)

end.

The output should be:

program noparmTest (input, output);

var a, b : integer;

c : real;

function two:

var n : integer;

begin

c = a + b;

two = -c

end

begin

a = 1;

b = 2;

c = a+b+two;

write(c)

end.

Here is a more challenging test file (which happens not to be legal Pascal):

program badPascal ( input, output );

var x, y, z : integer; {single colon is OK}

begin

if x = y then {will this mess up?}

x := y+ z; {should be no problem}

{check that blank line}

:= x\*y/12.3; {what about := at the start?}

x := y; y := z; p := q {multiple := in this line}

end.

To make the program easier, you could assume that there is only one := per line. Otherwise, you will need nested while loops.

Hints: use a Scanner and nextLine()for input. The outer while loop of the program uses hasNextLine() to decide if it should continue. Use indexOf() to look for :=. Use substring() and concat() (or +) to create the transformed line.

[Click here](http://www.webmastercert.com/moodle/index.html#29)to go back to the main menu.

## Exercise 2 — Lint Filter

An if statement or a while statement in Java should usually not contain the character =, unless it is part of ==, <=, >=, or !=. Sometimes, however, a single = is correct. Write a program that reads in a Java source program line by line and outputs only those lines that start with "if" or "while" (possibly preceded by whitespace) and that somewhere contain a single equal sign with at least one space on the left and right: " = " . The user can then inspect each line of the output for problems. A much better program would detect a single "=" even when not surrounded by spaces, but that is much harder to write.

Use the methods startsWith(), trim(), and indexOf(). There are other methods that would simplify this program, but don't use them here.

C:\>java Line < BuggyProgram.java

A program, such as this, that filters out suspicious lines is sometimes called a "lint filter". Here is a program to test your filter:

import java.util.Scanner;

import java.io.\*;

class BigProblems

{

public static void main ( String[] args )

{

boolean goOn = true;

Scanner scan = new Scanner( System.in );

System.out.println( "Enter an integer: ");

int data = scan.nextInt();

while ( goOn = true )

{

if ( data%2 = 1 )

System.out.println( "Your number is odd" );

if ( data%2=0 )

System.out.println( "Your number is even" );

System.out.println( "Enter another integer or 0 to quit: ");

data = scan.nextInt();

if ( data == 0 ) goOn = false;

}

}

}

[Click here](http://www.webmastercert.com/moodle/index.html#29)to go back to the main menu.

## Exercise 3 — File Compressor

The English, words "a" and "the" can mostly be removed from sentences without affecting the meaning. This is an opportunity for compressing the size of text files! Write a program that inputs a text file, line-by-line, and writes out a new text file where each line has the useless words eliminated.

First write a simple version of the program that replaces substrings " a " and " the " in each line with a single space. This will remove many words, but sometimes these words occur at the beginnings or ends of lines, and sometimes the words start with capitals. So, improve your first program so that it handles those situations as well.

C:\>java Remover < verbose.txt > terse.txt

Note: there are various replace() methods of class String that would simplify this program. Try to write this program without using them.

[Click here](http://www.webmastercert.com/moodle/index.html#29)to go back to the main menu.

## Exercise 4 — Secret Code

A text message has been encoded by replacing each character of the message with an integer. Each integer is an index into a key-phrase that contains all the lower case letters of the alphabet as well as the space character. The key-phrase may contain the same character in several locations. The encoded text is series of integers, like this:

35 10 10 33 9 24 3 17 41 8 3 20 51 16 38 44 47 32 33 10 19 38 35 28 49

To decode the message, look up each integer in the key-phrase and output the corresponding character. For example, say that the key-phrase is this (the index of each character has been written above it):

111111111122222222223333333333444444444455

0123456789012345678901234567890123456789012345678901

six perfect quality black jewels amazed the governor

using each integer from the encoded text as an index into the phrase results in the decoded message:

attack the bridge at dawn

Write a program that decodes a secret message contained in a text file. The first line of the text file contains the key-phrase. Then the file contains a sequence of integers, each of which indexes the key-phrase. Find the character corresponding to each integer and output the secret message. Note if a character character such as 'e' occurs several places in the key-phrase it may be encoded as different integers in different parts of the secret message.

(The recipient of the secret message gets only the file of integers and must put the key-phrase at the top of the file.) For example, here is the contents of a secret message file ready for the program:

six perfect quality black jewels amazed the governor

35 10 10 33 9 24 3 17 41 8 3 20 51 16 38 44 47 32 33 10 19 38 35 28 49

Here is a sample run of the program:

C:\> java Decode < secretFile.txt

attack the bridge at dawn

You will need the charAt() method of String.

Here is another secret message file, with key-phrase inserted, that you can use to test your program:

six perfect quality black jewels amazed the governor

31 16 2 3 4 42 48 7 27 9 10 43 12 13 35 15 1 40 18 3

20 15 33 23 24 32 26 29 28 27 21 31 25 14 34 14 36

42 38 19 40 41 27 3 44 50 46 42 48 49 50 6

**Defining Your Own Classes**

The previous chapters have discussed how to use objects and their methods. The objects were constructed using classes from the Java libraries. This chapter discusses how to define your own classes and construct objects using them.

**Chapter Topics:**

* Object Oriented Programming
* Syntax of Class Definitions
* Syntax of Method Definitions
* The return statement
* The void return type
* Object Oriented "Hello World" example
* Classes for Testing

**QUESTION 1:**

(Reviewsmile what is the difference between a class and an object?

**Answer:**

* A *class* is a description of a possible object.
* An *object* is a unique instance of a class.

Or you could say that a class is a plan for an object and an object is what results when the plan has been carried out.

**Object Oriented Programming**

Object-Oriented programming consists of three stages:

1. Creating the Program.
   * The programmer defines classes that describe objects that the program will use when it is running.
   * The programmer defines a class that contains the static main() method which is used to start the program running.
2. Compiling the Program.
   * The program is compiled into bytecode.
3. Running the Program.
   * The java interpreter looks for a static main() method and starts running it.
     + Since main() is static it can start running even though no objects have been created, yet.
   * As the program runs, objects are created and their methods are activated.
     + The program does its work by creating objects and activating their methods.
     + The exact order of object creation and method activation depends on to task to be performed and the input data.

The stage where the compiled program is running is called **run time**. Your program is like the script of a play. The work that you describe in your program (or play) is performed after you have finished the description and put it in the hands of the CPU (or actors). At run time (show time), Things Happen. Your beautiful program comes to life. Data is read in; calculations are performed; files are written.

**QUESTION 2:**

Object Oriented Programming is like assembling a team of human workers to get a job done:

1. First you plan on the workers and material you will need.
2. Then you assembly your workers and material.
3. Then the team works with the material until the job is done.

(Thought questionsmile You have decided to open a pizza delivery business consisting of yourself and several employees. What job descriptions will these employees have?

**Answer:**

You probably need the following:

* A boss (you, of course!)
* Cooks.
* Order-takers.
* Delivery people.

**Object Oriented Pizzas**

Maybe you need a few other job classifications. Perhaps there should be an order handler that takes the pizza from the cook and prepares it for delivery. Or maybe that should be the order-taker's job?

In thinking these thoughts you are doing something close to *object oriented design*. It is not that easy to get right, even for familiar situations. If you were designing your pizza delivery business you would probably spend several weeks (or months) designing your business plan. When you have the final design, you hire employees and buy pizza-making material.

Then it is run time.

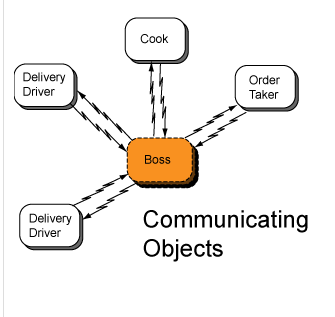
**QUESTION 3:**

Once your pizza delivery business is up and running, will your employees need to communicate with each other?

### Answer:

Of course. To get the work done, humans (and objects) need to ask each other to do their specialized tasks and need to give each other information necessary for the jobs.

# Communicating Objects



A running program is a collection of objects that are each doing their own task and communicating with other objects. The picture shows a running program (or a pizza shop). The jagged lines represent communication. The boss is special and so is drawn with dotted lines.

In Java programs the main() method is special because it is where the Java interpreter starts the whole program running. The main() method is a static method, which means that there will be only one instance of it and that it exists (as part of a class) before any objects have been created.

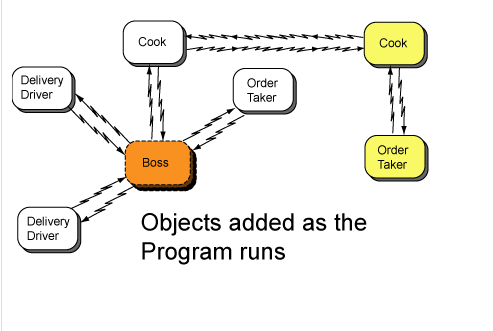
### QUESTION 4:

Say that after a few days your object-oriented pizza business gets more orders than it can handle. What will you do?

### Answer:

Add more employees.

# More Objects Created at Run Time



Object-oriented programs can create more objects as the program runs. The new objects might be needed to handle additional data that the program needs to work with. The picture shows your business after a few more employees have been added.

Notice that objects (workers) can communicate with each other, not just with main() (the boss). Some objects do not communicate with main() at all, but communicate only with other objects in the system.

### QUESTION 5:

(Reviewsmile Can a class description be used to create more than one object?

### Answer:

Yes. A big program might need thousands of objects as it runs, but might have only a few dozen class descriptions.

# Syntax of a Class Definition

Class definitions look like this:

class ClassName

{

Descriptions of the instance variables and methods each object

will have and the constructors that initialize a new object.

}

Often programmers separate the definition into three sections:

class ClassName

{

// Description of the variables.

// Description of the constructors.

// Description of the methods.

}

Separating the class into sections is done for clarity. It is not a rule of the language. A simple class might have just a few variables and be defined in just a few lines of code. A large, complicated class might take thousands of lines of code for its definition.

### QUESTION 6:

Does each object require a main() method?

**Answer:**

No. You need only one main() method for the Java virtual machine to use in starting your program.

**A Tiny Example**

class HelloObject

{

// method definition

void speak()

{

System.out.println("Hello from an object!");

}

}

class HelloTester

{

public static void main ( String[] args )

{

HelloObject anObject = new HelloObject();

anObject.speak();

}

}

Above is a complete program which includes two class definitions. The definition of class HelloObject includes a method but no instance variables, so objects of class HelloObject have no instance variables. The class does have a constructor but it is not explicitly defined in the code (this will be discussed further).

The definition of class HelloTester contains only the static main() method. When the main() method starts, it constructs a HelloObject and then invokes that object's speak() method.

Remember that name of the file must match the name of the class that contains the main() method. (Also remember that upper and lower case are important both in the file name and in the class name.) Here is an example run:

C:\>javac HelloTester.java

C:\>java HelloTester

Hello from an object!

C:\>

**QUESTION 7:**

* What two classes are defined in this code?
* What method is defined in the first class?

**Answer:**

* What two classes are defined in this code?
  + HelloObject and HelloTester
* What method is in the first class?
  + A HelloObject object has a speak() method.

**Answer:**

* What two classes are defined in this code?
  + HelloObject and HelloTester
* What method is in the first class?
  + A HelloObject object has a speak() method.

**Method Definition**

Method definitions look like this:

returnType methodName( parameterList )

{

// *Java statements*

**return** returnValue;

}

The returnType is the type of value that the method hands back to the caller of the method. Methods in classes you define can return values just as do methods from library classes. The **return**statement is used to hand back a value to the caller.

If you want a method that does something, but does not return a value to the caller, use a return type of **void** and do not use a return value with the **return** statement. The **return** statement can be omitted; the method will automatically return to the caller after it executes. Here is the method from the example program:

// method definition

void speak()

{

System.out.println("Hello from an object!");

}

**QUESTION 8:**

Examine the program.

Is a HelloTester object created when the program runs?

**Answer:**

* Is a HelloTester object created when the program runs?
  + No.
  + The static main() method does not need an object to run.

**Testing Class**

The Java interpreter starts a program by looking for a static main() method inside of the HelloTester.class file. Since the method is static the interpreter can run it without first constructing an object.

It is convenient to have a separate class that serves no other purpose than to contain the main() method. This testing class is used to start things running. Usually its main() constructs objects of various classes and calls their methods. These objects do the real work of the program.

The source file for the program is named HelloTester.java. When you compile the file, the compiler outputs two separate files of bytecodes, one for each class:

C:\chap30>javac HelloTester.java

C:\chap30>dir

11/13/98 10:07p 257 HelloTester.java

11/13/98 10:40p 476 HelloObject.class

11/13/98 10:40p 373 HelloTester.class

3 File(s) 1,106 bytes

To run the program, type:

java HelloTester

The Java interpreter finds the main() method in the HelloTester class and starts it running.

**QUESTION 9:**

When the Java interpreter needs the definition HelloObject, where will it be found?

### Answer:

In the file of bytecodes, HelloObject.class.

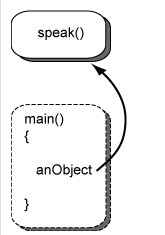
### QUESTION 10:

Could you activate the speak() method without creating an object?

### Answer:

No.

# Picture of the Program



The speak() method is part of an object. It exists only when there is an object to contain it (unlike a static method that does not need an object). The picture shows the action, just as step 3 starts.

The variable anObject is a reference to the object that was constructed. That object contains the method speak(). So main() can find and activate speak() with:

anObject.speak();

You might think that this is a needlessly complicated way to write a message on the monitor. And you would be correct. But remember that this is a simple example of object oriented programming. Later on, when the task gets really complicated, using objects greatly simplifies programming. Writing large programs in Java takes only one third to one fifth of the time it takes to write in some other languages.

### QUESTION 11:

Does the class HelloObject have a constructor?

### Answer:

Yes. To construct an object, there must be a constructor.

# Default Constructor

Here is where the constructor is used in the main() method:

HelloObject anObject = new HelloObject(); // 2. a HelloObject

// is created using its

// default constructor

But here is the class definition of HelloObject:

class HelloObject

{

void speak()

{

System.out.println("Hello from an object!");

}

}

There is no constructor described in the class definition. This is OK. A class will have a constructor even if one is not explicitly included in the class definition. If the class definition does not include a constructor a **default constructor** is automatically supplied by the Java compiler.

### QUESTION 12:

So, it looks like the default constructor does no work. Is this correct?

**Answer:**

No.

**Work Behind the Scenes**

The default constructor does a great deal of work "behind the scenes" when it is used. It works with the Java virtual machine to find main memory for the object, sets up that memory as an object, puts in it the variables and methods specified in the class definition, and returns an object reference to your program. All of this is quite complicated, and it is fortunate that you don't have to write code for it.

If you need to initialize some variables in a newly constructed object, then need to write a constructor as part of a class definition. In the constructor, all you need to do is write the statements that initialize the variables. The "behind the scenes" work still is included automatically.

**Syntax Rule:** If you define one or more constructors for a class, then those are the only constructors that the class has. The default constructor is supplied automatically only if you define no constructors.

This rule actually makes sense. If you are defining constructors for a class, then it is likely that you want to have control over the process and don't want an extra default constructor.

**QUESTION 13:**

(Design Question: ) In our design so far, every HelloObject object prints the same string to the monitor. How could this class be improved?

### Answer:

It would be nice if different objects printed different things.

# Methods can use Variables

If different instances (different objects) of the HelloObject class print different strings, then each instance must have its own data, which must be initialized. The class definition needs a constructor. Here is what a class definition usually looks like:

class ClassName

{

Description of the variables.

Description of the constructors.

Description of the methods.

}

**Important:** An object's methods use that object's instance variables. Remember that an object has identity, state, and behavior. "Identity" means that each object has its own variables. The "state" of an object is the values held in its variables. The "behavior" of an object is its methods, which use the object's own variables.

### QUESTION 14:

If each instance of HelloObject had its own message, where would that message be kept?

### Answer:

A reference to the message is kept in one of the object's variables.

# Improved Class

Here is what we have so far for HelloObject:

class HelloObject

{

void speak()

{

System.out.println("Hello from an object!");

}

}

Here is a start on improving HelloObject:

Top of Form

class HelloObject

{

 ; // reference variable for the object's message

void speak()

{

System.out.println(  ); // print the object's message

}

}

Bottom of Form

### QUESTION 15:

Modify the definition of HelloObject to include a String reference variable. Modify the speak() method so that it uses that variable.

### Answer:

The near-complete class is below.

# Using a Constructor

Here is the class with a String reference variable included. Of course, you may have used a different name for it.

class HelloObject

{

String greeting;

void speak()

{

System.out.println( greeting );

}

}

The class is not complete because there is no way to initialize the greeting (we will get to this shortly). It would be ince if the object could used like this:

class HelloTester

{

public static void main ( String[] args )

{

HelloObject anObject = new HelloObject("A Greeting!");

anObject.speak();

}

}

### QUESTION 16:

Where in the above code is a constructor being used? What is its parameter?

**Answer:**

Where in the above code is a constructor being used?

HelloObject anObject = new HelloObject("A Greeting!");

------------------------------

What is its parameter? A reference to the string, "A Greeting!"

**Constructor Definition Syntax**

The class needs a *constructor*. Constructor definitions look like this:

className( parameterList )

{

*Statements involving the variables of the*

*class and the parameters in the parameterList.*

}

No return type is listed in front of *className*. The return type is automatically a reference to an object of the class. There is no return statement in the body of the constructor. A reference is returned automatically. (There would be no reason to have a constructor, otherwise.) The constructor has the same name as the class. The *parameterList* is a list of values and their types:

TypeName1 parameterName1, TypeName2 parameterName2, ... as many as you need

It is OK to have an empty parameter list. A class often has several constructors with different parameters. Each one builds the same class of object, but the different constructors use different sources of data for object initialization.

Usually the method that invokes a constructor saves the returned reference in a variable. But sometimes an object is constructed for temporary use and its reference is not saved. The object is used once for some brief purpose and then becomes garbage.

**QUESTION 17:**

Can a parameter be an object reference?

Can a parameter be a primitive data type?

### Answer:

Yes to both.

# Coding a Constructor

Here is HelloObject with an unfinished constructor. In this constructor the parameter is an object reference.

Top of Form

class HelloObject

{

String greeting;

HelloObject( String st )

{

 =  ;

}

void speak()

{

System.out.println( greeting );

}

}

Bottom of Form

Examine the parameter list of the constructor:

String st

This says that the constructor is given a reference to a String when it is used (when it is called). The name of the parameter is st. In the body of the constructor, st represents the data. The constructor will initialize the variable greeting with data that is supplied when the constructor is used. For example, in the main() method above the String "A Greeting!" is supplied as data to the constructor.

### QUESTION 18:

Fill in the blank so that the constructor is complete.

### Answer:

A complete program is below.

# Completed Constructor

Here is a complete program. The constructor in the definition of HelloObject has a parameter named st. This parameter is used to pass data into the constructor. The constructor itself does nothing but copy it to the variable greeting of the object.

class HelloObject

{

String greeting;

HelloObject( String st )

{

greeting = st;

}

void speak()

{

System.out.println( greeting );

}

}

class HelloTester

{

public static void main ( String[] args )

{

HelloObject anObject = new HelloObject("A Greeting!");

anObject.speak();

}

}

### QUESTION 19:

What will this program print on the monitor when it is run?

### Answer:

When run, the program will print:

A Greeting!

to the monitor.

# Steps in Program Execution

Here is how the program does this:

class HelloObject // 3a. the class definition is

{ // used to construct the object.

String greeting;

HelloObject( String st ) // 3b. the constructor is used to

{ // initialize the variable.

greeting = st;

}

void speak() // 4a. an object has its own copy

{ // of this method.

System.out.println( greeting ); // 4b. the object's method uses

} // the data in its variable

} // greeting.

class HelloTester

{

public static void main ( String[] args ) // 1. main starts running

{

HelloObject anObject =

new HelloObject("A Greeting!"); // 2. the String "A Greeting"

// is constructed.

// 3. A reference to the String is

// passed to the HelloObject

// constructor.

// A HelloObject is created.

anObject.speak(); // 4. the object's speak()

// method is activated.

}

}

You usually don't think about what is going on in such detail. But you should be able to do it when you need to.

Notice that a String object containing "A Greeting!" is constructed before the HelloObject constructor is even called. Strings are objects (of course) so they must be made with a constructor. Remember that Strings are special because an object can be constructed without using the new operator. This is what the literal "A Greeting!" does.

### QUESTION 20:

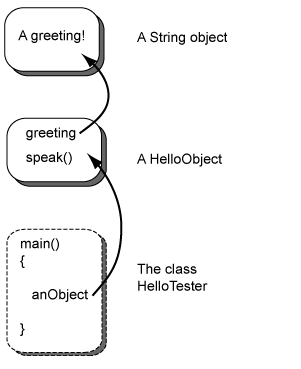
Top of Form

(Trick Questionsmile How many objects exist just before this program stops running? Click here for a

### Answer:

Two: the String object "A Greeting!" and the HelloObject.

# Picture of the Running Program



Here is a picture showing the objects in the program just before it stops running. Of course, when it stops running all the objects become inactive and are reclaimed by the garbage collector.

While the program is running, there is a way to find every object:

1. The main method can find the HelloObject through the reference in the variable anObject.
2. The main method activates the speak() method by anObject.speak().
3. The speak() method can find the String object using the variable greeting, which is part of the HelloObject object.
4. System.out.println(greeting) in that method writes out the data in the String object.

Don't worry terribly much about all these details. Look them over a few times and then move on. Come back to this chapter in a few days when things have had a chance to soak in.

### QUESTION 21:

(Obvious questionsmile Can there be several HelloObject objects created in the main() method?

### Answer:

Of course. The class definition for HelloObject works like a cookie cutter. We can cut out as many cookies as we want.

# Expanded Program

Here is an expanded program that creates several HelloObjects (and several String objects.)

class HelloObject

{

String greeting;

HelloObject( String st )

{

greeting = st;

}

void speak()

{

System.out.println( greeting );

}

}

class HelloTester

{

public static void main ( String[] args )

{

HelloObject object1 = new HelloObject("Mercury");

HelloObject object2 = new HelloObject("Venus");

HelloObject object3 = new HelloObject("Earth");

HelloObject object4 = new HelloObject("Mars");

object1.speak();

object2.speak();

object3.speak();

object4.speak();

}

}

The program writes this to the monitor:

Mercury

Venus

Earth

Mars

### QUESTION 22:

What names may be used for parameters?

### Answer:

The same names as can be used for variables (and methods and classes.)

# this

Parameter names follow the same rules as the other names you might pick for a program. Names of this type are called identifiers. The rules for identifiers are given in chapter 9. Here is an altered definition of our HelloObject class:

class HelloObject

{

String greeting;

HelloObject( String greeting )

{

this.greeting = greeting;

}

void speak()

{

System.out.println( greeting );

}

}

The parameter is named greeting, a legal and sensible name. But the instance variable is also named greeting. This is OK, but can lead to some confusion. To avoid confusion, use the reserved word this to show when an identifier refers to an object's instance variable.

### QUESTION 23:

What type of constructor would a class named Boa have?

### Answer:

A Boa Constructor.

# Designing a Class  (Miles per Gallon)

This chapter contains another example of designing a class, and then using objects of that type. The class is named Car, and is a simple miles-per-gallon calculator.

#### Chapter Topics:

* How classes are documented
* Writing a class definition
* Instance variables
* Using instance variables in a method
* Constructors
* Constructor parameter lists

### QUESTION 1:

What three values are you likely to use in calculating miles per gallon for a car?

### Answer:

You would expect to use:

1. Starting odometer reading,
2. Ending odometer reading, and
3. Gallons of gas used between the readings.

# Specifications for the Car class

Think about the classes you need before you start writing. This makes programming easier and your programs will have fewer bugs. **Object oriented design** means deciding what classes you need, what data the objects hold, and how the objects will behave. Let us do that with the Car class.

**Car**

A class that calculates miles per gallon.

**Variables**

* double startMiles; // Starting odometer reading
* double endMiles; // Ending odometer reading
* double gallons; // Gallons of gas used between the readings

**Constructors**

* Car( double startOdo, double endingOdo, double gallons )  
  Creates a new instance of a Car object with the starting and ending odometer readings and the number of gallons of gas consumed.

**Methods**

* double calculateMPG()   
  calculates and returns the miles per gallon for the car.

Look at the parameter list for the constructor:

Car( double startOdo, double endingOdo, double gallons );

This says that the constructor must be called with three items of data: three double precision values.

### QUESTION 2:

Could a main() method create a Car object?

### Answer:

Sure, as long as it has access to the class definition.

# Using a Car Object

To see if the design for Car is what you want, use it in a program. If Car is designed well, writing the program should be easy.

import java.util.Scanner ;

import Car;

class MilesPerGallon

{

public static void main( String[] args )

{

Scanner scan = new Scanner(System.in);

double startMiles, endMiles, gallons;

System.out.print("Enter first reading: " );

startMiles = scan.nextDouble();

System.out.print("Enter second reading: " );

endMiles = scan.nextDouble();

System.out.print("Enter gallons: " );

gallons = scan.nextDouble();

Car car = new Car( , ,  );

System.out.println( "Miles per gallon is " + car.calculateMPG() );

}

}

User interaction is done just as in previous programs. All input values are double precision.

### QUESTION 3:

Fill in the blanks so that the Car constructor uses data from the user.

### Answer:

The complete program is given below.

# Finished Program

Here is the complete main(). Unfortunately, until you write the code for Car the program can't be run.

import java.util.Scanner ;

import Car;

class MilesPerGallon

{

public static void main( String[] args )

{

Scanner scan = new Scanner(System.in);

double startMiles, endMiles, gallons;

System.out.print("Enter first reading: " );

startMiles = scan.nextDouble();

System.out.print("Enter second reading: " );

endMiles = scan.nextDouble();

System.out.print("Enter gallons: " );

gallons = scan.nextDouble();

Car car = new Car( startMiles, endMiles, gallons );

System.out.println( "Miles per gallon is " + car.calculateMPG() );

}

}

### QUESTION 4:

We still need a definition for class Car. Can a programmer write a definition for the class Car?

### Answer:

Of course!

# Class Definition

Here is the miles per gallon program, now with a skeleton for the Car class. To keep the program short, user interaction has been left out.

class Car

{

// instance variables

// constructor

// methods

}

class MilesPerGallon

{

public static void main( String[] args )

{

Car car = new Car( 32456, 32810, 10.6 );

System.out.println( "Miles per gallon is " + car.calculateMPG() );

}

}

The source file must be named MilesPerGallon.java after the name of the class that contains main().

### QUESTION 5:

Decide what variables should go in the data section. Look back to the [Car class](http://www.webmastercert.com/moodle/mod/book/ch31_2.html#Car,%20class) for the class to see what you need.

### Answer:

The data of a Car object should be:

1. Stating odometer reading,
2. Ending odometer reading, and
3. Gallons of gas used between the readings.

The names of the variables are up to the programer.

# Filling in the Definition

Here is the program with some of the Car definition filled in:

class Car

{

// instance variables

double startMiles; // Stating odometer reading

double endMiles; // Ending odometer reading

double gallons; // Gallons of gas used between the readings

// constructor

// methods

}

class MilesPerGallon

{

public static void main( String[] args )

{

Car car = new Car( 32456, 32810, 10.6 );

System.out.println( "Miles per gallon is " + car.calculateMPG() );

}

}

An **instance variable** is a variable that holds part of the state of an object. Each object (each "instance" of the class) contains its own instance variables. Instance variables hold on to their values as long as the object exists. An assignment statements can change the value in an instance variable (see the next chapter), but otherwise it holds its value for the lifetime of the object.

### QUESTION 6:

What must the constructor of Car be named?

### Answer:

The constructor must be named Car, the same as the name of the class.

# Constructor Parameter List

The constructor is used when an object is constructed. Most of the work in constructing an object is automatically done by the Java system. Usually you need to write only a few statements that initialize a instance variables. The constructor for Car will initialize the three variables of the object. Here is the class definition, with the constructor partially finished:

class Car

{

// instance variables

double startMiles; // Stating odometer reading

double endMiles; // Ending odometer reading

double gallons; // Gallons of gas used between the readings

// constructor

Car(  ,  ,   )

{

}

// methods

}

The parameter list of a constructor looks like this:

dataType parameterName , dataType parameterName , and so on

Each dataType is the type of one item of data that will be handed to the constructor, and each parameterName is a name that is used for that item of data.

### QUESTION 7:

Fill in the blanks in the parameter list. The data type of each parameter should match the data type of an instance variable. The names of the parameters are up to you.

### Answer:

The completed parameter list is seen below.

# Initializing the Data

In this class, the parameter list uses three parameters to match the three instance variables that will be initialized.

class Car

{

// instance variables

double startMiles; // Stating odometer reading

double endMiles; // Ending odometer reading

double gallons; // Gallons of gas used between the readings

// constructor

Car( double first, double last, double gals )

{

 = first;

 = last;

 = gals;

}

// methods

}

class MilesPerGallon

{

public static void main( String[] args )

{

Car car = new Car( 32456, 32810, 10.6 );

System.out.println( "Miles per gallon is " + car.calculateMPG() );

}

}

**Note:** In most classes there is NOT an exact match between the parameters of a constructor and the instance variables. There may be fewer (or more) parameters than instance variables. Also, the constructor's parameters do not have to be in the same order as the instance variables (this might help keep things straight, but the compiler does not care).

### QUESTION 8:

Now fill in the assignment statements to complete the constructor.

### Answer:

The complete constructor is seen below.

# Complete Constructor

class Car

{

// instance variables

double startMiles; // Stating odometer reading

double endMiles; // Ending odometer reading

double gallons; // Gallons of gas used between the readings

// constructor

Car( double first, double last, double gals )

{

startMiles = first;

endMiles = last;

gallons = gals;

}

// methods

double calculateMPG()

{

return  ;

}

}

class MilesPerGallon

{

public static void main( String[] args )

{

Car car = new Car( 32456, 32810, 10.6 );

System.out.println( "Miles per gallon is " + car.calculateMPG() );

}

}

For many programs all that the constructor does is to copy values from its parameters to the instance variables of the new object. You might wonder why you need to do this. Why not just leave the data in the parameters? There are two reasons:

1. The constructor's parameters can be "seen" only by its own statements. A method such as calculateMPG() cannot see the parameters of the constructor.
2. Data in parameters is temporary. Parameters are used to communicate data, not to hold data.

Think of a parameter as a scrap of paper containing information handed to the constructor. The constructor has to copy the information to someplace permanent that can be seen by the other methods.

### QUESTION 9:

Top of Form

Now complete the calculateMPG() method by filling in the blank.

### Answer:

The complete program is below. Copy it to a file and run it.

# Complete Program

Here is the complete program. The calculateMPG() method returns a double value to the caller. Since it returns a value, there must be a return statement within its body that returns a value of the correct type.

In the complete program, the calculateMPG() method uses the instance variables of its object to calculate miles per gallon.

class Car

{

// instance variables

double startMiles; // Stating odometer reading

double endMiles; // Ending odometer reading

double gallons; // Gallons of gas used between the readings

// constructor

Car( double first, double last, double gals )

{

startMiles = first ;

endMiles = last ;

gallons = gals ;

}

// methods

double calculateMPG()

{

return (endMiles - startMiles)/gallons ;

}

}

class MilesPerGallon

{

public static void main( String[] args )

{

Car car = new Car( 32456, 32810, 10.6 );

System.out.println( "Miles per gallon is " + car.calculateMPG() );

}

}

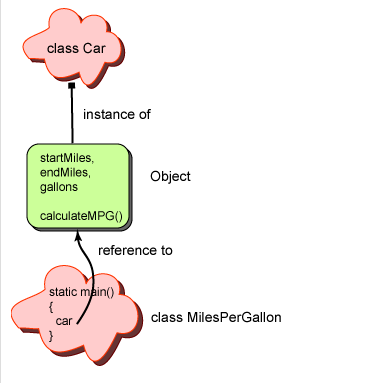
### QUESTION 10:

* How many objects are in the system the very instant the program starts running?
* How many objects are in the system just before the program stops running?

### Answer:

* How many objects are in the system the very instant the program starts running?
  + Zero.
  + (This is always the answer to this question.)
* How many objects are in the system just before the program stops running?
  + One, the object referenced by car.
  + (The String object in the println has already become garbage.)

# Picture of Main Memory



When a program starts running there are no objects, just class definitions and a static main() method. Usually main() then constructs some objects and calls their methods to do the work of the program. In our example, main() constructs just one object and then calls its calculateMPG() method.

The picture shows the variable car in the static main() method referring to the object that has been constructed following the class definition of Car. The static main() method is part of theMilesPerGallon class.

In the picture, the class definitions are shown as pink clouds as a reminder that they are not full objects. The object in this picture is shown as a solid rectangle.

### QUESTION 11:

Could several objects of type Car be constructed?

### Answer:

Sure.

# Two Cars

Here is the program with a few more blank lines:

class Car

{

. . . .

}

class MilesPerGallon

{

public static void main( String[] args )

{

Car car1 = new Car( 32456, 32810, 10.6 );

System.out.println( "Miles per gallon of car 1 is " + car1.calculateMPG() );

Car car2 = new Car( , ,  );

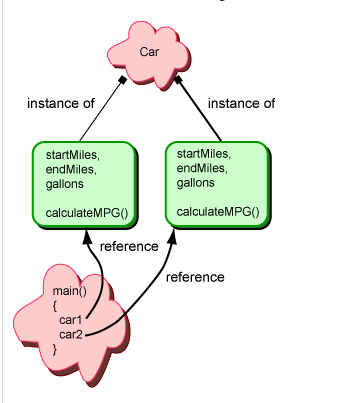
System.out.println( "Miles per gallon of car 2 is " + car2.calculateMPG() );

}

}

### QUESTION 12:

Fill in the blanks for the second car, which has

* A starting odometer reading of 100000
* An ending odometer reading of 100300
* Gas consumption of 12.5 gallons.
* **Answer:**
* Easy. The filled in program is below.
* **Picture of Two Objects**
* 
* In this program, two defferent Car objects are constructed, each with different data, and the miles per gallon calculation is performed on each.
* class Car
* {
* . . . .
* }
* class MilesPerGallon
* {
* public static void main( String[] args )
* {
* Car car1 = new Car( 32456, 32810, 10.6 );
* System.out.println( "Miles per gallon of car 1 is "
* + car1.calculateMPG() );
* Car car2 = new Car( 100000, 100300, 10.6 );
* System.out.println( "Miles per gallon of car 2 is "
* + car2.calculateMPG() );
* }
* }
* The picture shows the situation just after the second object has been constructed. Two objects have been constructed according to the definition of Car. The definition of class MilesPerGalloncontains the static main() method.
* **QUESTION 13:**
* There are two objects, but each uses the same names for its instance variables! Is this a mistake?

### Answer:

No. Each object has its own identity, so there is no confusion about which variables are which. Each object's constructor gave that object's instance variables the correct values.

# Slightly Different Program

If this is confusing, remember that object oriented programming is supposed to imitate the real world. Think of some objects of the real world, say objects of the class Human. Each object has its own identity (ie. Bob is a different individual from Jill) even though each has parts that have the same name. It is not confusing to talk of "Bill's heart" and "Bill's nose," and "Jill's heart" and "Jill's nose." With "dot notation" this would be Bob.heart, Bob.nose, Jill.heart, and Jill.nose.

Below is a slightly different version of the program.

class Car

{

. . . .

}

class MilesPerGallon

{

public static void main( String[] args )

{

Car car = new Car( 32456, 32810, 10.6 );

System.out.println( "Miles per gallon of car 1 is "

+ car.calculateMPG() );

car = new Car( 100000, 100300, 10.6 );

System.out.println( "Miles per gallon of car 2 is "

+ car.calculateMPG() );

}

}

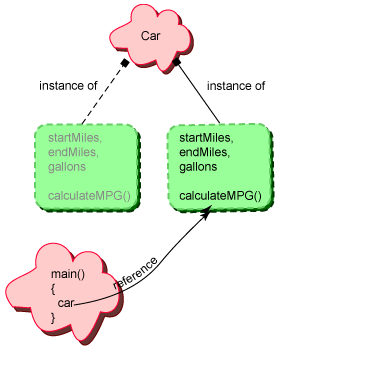
### QUESTION 14:

How does this program differ from the previous program?

### Answer:

A single reference variable car is used in turn for each object.

# Junked Car



The output to the screen of this program is the same as the previous. However, since the reference variable car is used when the second Car is constructed, the first car becomes garbage before the end of the program.

Car car = new Car( 32456, 32810, 10.6 );

System.out.println( "Miles per gallon of car 1 is "

+ car.calculateMPG() );

car = new Car( 100000, 100300, 10.6 );

System.out.println( "Miles per gallon of car 2 is "

+ car.calculateMPG() );

The picture shows the situation just after the second car is constructed. The first car no longer has a reference to it, so it is now garbage.

### QUESTION 15:

(Review Questionssmile

1. Do the instance variables of an object hold values for the lifetime of the object?
2. Do the parameters of a constructor hold values for the lifetime of the object?

### Answer:

1. Do the instance variables of an object hold values for the lifetime of the object?
   * Yes—an object is a "thing" that has state (its unique characteristics.) The state is kept in instance variables.
2. Do the parameters of a constructor hold values for the lifetime of the object?
   * No—the parameters of a constructor are part of a temporary "message" to the constructor. After the parameters have been used, they are gone.

# Possible Errors

Of course, the data in the "message" to the constructor will usually be stored in the instance variables, where it will remain until the object is destroyed (or until the instance variables are deliberately changed).

Here is an interesting program. Is anything wrong?

class Car

{

// instance variables

double startMiles; // Stating odometer reading

double endMiles; // Ending odometer reading

double gallons; // Gallons of gas used between the readings

// constructor

Car( double first, double last, double gals )

{

startMiles = first ;

endMiles = last ;

gallons = gals ;

}

// methods

double calculateMPG()

{

return (last - first)/gals ;

}

}

class MilesPerGallon

{

public static void main( String[] args )

{

Car car = new Car( 32456, 32810, 10.6 );

System.out.println( "Miles per gallon is " + car.calculateMPG() );

}

}

### QUESTION 16:

Top of Form

Examine the program. Is there anything wrong?

### Answer:

The calculateMPG() method can not use the parameters of the constructor.

# Can't Use Constructor's Parameters in a Method

In fact, the calculateMPG() method can not even see the parameters of the constructor. Another way of saying this is that the **scope** of the parameters is limited to the body of the method. The compiler will complain that first, last, and gals are "undefined variables" because they are used outside of their scope. Look back to the [complete program](http://www.webmastercert.com/moodle/mod/book/view.php?id=5490&chapterid=1254#program,%20Car%20class) to see the correct method.

class Car

{

// instance variables

double startMiles; // Stating odometer reading

double endMiles; // Ending odometer reading

double gallons; // Gallons of gas used between the readings

// constructor

Car( double first, double last, double gals )

{

startMiles = first ;

endMiles = last ;

gallons = gals ;

}

// methods

double calculateMPG()

{

return (last - first)/gals ; // WRONG, WRONG, WRONG

}

}

class MilesPerGallon

{

public static void main( String[] args )

{

Car car = new Car( 32456, 32810, 10.6 );

System.out.println( "Miles per gallon is " + car.calculateMPG() );

}

}

### QUESTION 17:

Are you about out of gas?

### Answer:

Yes.

# Fill in the Blanks

# 

**Instructions:**   This is an ungraded fill-in-the-blank exercise. Each question consists of a sentence with one or two words left out. A button represents the missing word(s). For each question, think of the word or phrase that should fill each blank, then click on the buttons to see if you are correct. No grade is calculated for this exercise.

Top of Form

1.      **Design of the Telescope Class**.     The three most important numbers describing a telescope are: the diameter of the main lens (the one in front), the focal length of the main lens, and the focal length of the eyepiece. From these values other charactersitics of the telescope such as its magnification and the f-number of the main lens are calculated.

Fill in the blanks in the following design for the class:

**class** Telescope

A class that models a field telescope.

**Constructors**

Telescope ( double diameter ,

double mainLength,

double eyeLength )

**Methods**

// calculate the magnification of the telescope

double magnification()

// calculate the f-number of the telescope

double fNumber()

Bottom of Form

2.      **Checking the Design**.    To check the design, write a small program that uses the class to see if it works well. Write a program that creates a Telescope object with a main lens that has a diameter of 3.0 inches, a focal length of 6.5 inches, and an eyepiece focal length of 0.8 inches. Write out its magnification and f-number.

Top of Form

class TelescopeTester

{

public static void main ( String[] args )

{

Telescope tele = new Telescope(3.0,6.5,.8);

System.out.println( "Power: " + tele.magnification() +

" F-number: " + tele.fumber() );

}

}

Bottom of Form

3.      **Skeleton of the Class**.    Fill in the blanks that give the over-all design of the class.

Top of Form

class Telescope

{

// Instance Variables

// Constructors

// Methods

}

Bottom of Form

4.      **Fill in Instance Variables**.    Fill in the data type of each instance variable.

Top of Form

class Telescope

{

// Instance Variables

double diameter;

double mainLength;

double eyeLength;

// Constructors

// Methods

}

Bottom of Form

5.      **Complete the Constructor**.    The constructor will initialize the instance variables of the object being constructed.

Top of Form

class Telescope

{

// Instance Variables

double diameter;

double mainLength;

double eyeLength;

// Constructors

Telescope( double diameter , double mainLength , double eyeLength )

{

this.diameter = diameter ;

this.mainLength = mainLength ;

this.eyeLength = eyeLength;

}

// Methods

}

Bottom of Form

When an instance variable and a parameter use the same identifier, you specify the instance variable of the object by saying "this.identifier" as in the above.

6.      **Complete a Method**.     The documentation for the magnification() method says it looks like this:

// calculate the magnification of the telescope

double magnification()

The formula to use is: magnification = mainLength/eyeLength

class Telescope

{

// Instance Variables

double diameter;

double mainLength;

double eyeLength;

// Constructors

Telescope ( double diameter, double mainLength, double eyeLength )

{

this.diameter = diameter ;

this.mainLength = mainLength ;

this.eyeLength = eyeLength ;

}

// Methods

double magnification()

{

return mainLength / eyeLength ;

}

}

You don't have to use "this" in the method because it is clear that the identifiers refer to the instance variables. The magnification() method can not even see the parameters of the constructor. (Remember, the parameters are like a private telegram to the constructor.)

7.      **Complete the other Method**.     The documentation for the says it looks like this:

// calculate the magnification of the telescope

double fNumber()

The formula to use is: fNumber = mainLength/diameter

Top of Form

class Telescope

{

// Instance Variables

double diameter;

double mainLength;

double eyeLength;

// Constructors

Telescope ( double diameter, double mainLength, double eyeLength )

{

this.diameter = diameter ;

this.mainLength = mainLength ;

this.eyeLength = eyeLength ;

}

// Methods

double magnification()

{

return mainLength / eyeLength ;

}

double fNumber()

{

return mainLength / diameter ;

}

}

Bottom of Form

8.      **Import the Class**.     At this point all the coding is done. Say that the small test program (class TelescopeTester) is in a file called TelescopeTester.java and that the complete definition of the Telescope class is in a file called Telescope.java. The code for TelescopeTester has to tell the compiler that it is using a class defined outside its file with an import statement:

Top of Form

import Telescope ;

class TelescopeTester

{

public static void main ( String[] args )

{

Telescope tele = new Telescope( 3.0, 6.5, .8 ) ;

System.out.println( "Power: " + tele.magnification() +

" F-number: " + tele.fNumber() );

}

}

Bottom of Form

9.      **Compile each File**.     Each of the two files must be compiled:

Top of Form

C:\MyFiles>javac Telescope.java

C:\MyFiles>javac TelescopeTester.java

Bottom of Form

10.      **Run the Program**.     The java bytecode interpreter must be started with the \*.class file that contains the main() method:

Top of Form

C:\MyFiles>java TelescopeTester

Bottom of Form

End of the Exercise. If you want to do it again, click on "Refresh" in your browser window. [Click here](http://www.webmastercert.com/moodle/index.html#31)to go back to the main menu.

**Due before the last day of class:**

Modify the Car class of the chapter by adding two methods:

* boolean gasHog()
  + evaluates to true if the miles per gallon is lower than 15.0.
* boolean economyCar()
  + evaluates to true if the miles per gallon is higher than 30.0.

The constructor and the calculateMPG() method remain unchanged. Each of these new methods should use the calculateMPG() to get the miles per gallon, not calculate it themselves. An if-else statement picks the correct boolean return value.

Put user interaction back into the main() method so the user enters values for each car. The main() method uses these additional methods to write a message to the user if the car is a gas hog or an economy car.

You might be tempted to make one of these common design errors:

1. Saving miles per gallon in an instance variable of the object along with startMiles, endMiles, and gallons.
   * This almost seems logical, but is a poor design. Don't keep a permanent copy of a value that can be easily calculated from data. The reason for this is that it adds complexity to the object, but offers little advantage.
2. Directly calculating miles per gallon inside each of the new methods.
   * It is usually best to do a particular calculation in a method, and to use it whenever the calculation is needed. Now if there is a bug in the calculation, or the calculation must be modified, there is only one place to look.

Here is a sample run of the program:

C:\>java Miles

Enter first reading: 10000

Enter second reading: 10400

Enter gallons: 10

Miles per gallon: 40

Economy Car!

**Upload your .zip file below...**

# Class Design Example  (Checking Account)

This chapter shows the design of the CheckingAccount class, a simple version of a checking account.

#### Chapter Topics:

* Requirements Analysis
* Design
* Implementation
* Scaffolding
* Testing
* Aliasing (Review)

### QUESTION 1:

(Thought Question: ) Could a checking account be represented by a software object?

### Answer:

The following seem reasonable to me. You might have thought of others.

* Account number
* Name of account holder
* Current balance

# Requirements for the State

A real checking account would contain a great deal more information, but for this example, the above is enough.

Next, think about what methods are needed. That is, what behavior does a checking account have?

### QUESTION 3:

Think of three behaviors a checking account should have.

### Answer:

The following seem reasonable to me:

* Accept a deposit
* Process a check
* Get the current balance

You might have thought of others, but the above will be enough for now.

# Requirements for Behavior

The constructor creates a new checking account and initializes it with its account number, the account holder's name, and the starting balance. Now we have a fair idea about what the class looks like:

* Data
  + Account number
  + Name of account holder
  + Current balance
* Constructor
  + Create the object; initialize the three data items
* Methods
  + Accept a deposit
  + Process a check
  + Get the current balance

For a large project, creating specifications is hard because objects interact with perhaps hundreds of other objects. In this example program, objects interact only with main().

### QUESTION 4:

Is there enough information to start writing the program?

**Answer:**

Not quite. We need to know more about how deposits and checks are processed.

**More Requirements**

The requirements describe what each method does. The method to accept a deposit adds an amount (in cents) to the current balance. The current balance can be negative or positive.

The method to process a check subtracts the amount of the check from the current balance. Overdrafts are allowed, so the balance can become negative. However, if the balance is less than $1000.00 before the check is processed, $0.15 is charged for each check.

To simplify the program, assume that all data is correct (so the methods do not check for errors).

It is time to think about design. The first thing to do is sketch out the class definition:

class CheckingAccount

{

*instance variables declarations*

*constructors*

*methods*

}

Although Java syntax does not require it, it is helpful to put different parts of a class definition in different sections, as above.

**QUESTION 5:**

Think of a reasonable variable name (identifier) and data type for each variable:

Top of Form

|  |  |  |
| --- | --- | --- |
|  | **Name** | **Type** |
| Account number. |  |  |
| Name of account holder. |  |  |
| Current balance. |  |  |

**Answer:**

|  |  |  |
| --- | --- | --- |
|  | **Name** | **Type** |
| Account number. | accountNumber | String |
| Name of account holder. | accountHolder | String |
| Current balance. | balance | int |

**Implementing the Constructor**

You might have thought of equally valid names. The account number should be a String because it is not expected to take part in arithmetic operations. Sometimes account numbers contain dashes or other non-digit characters. The balance is kept in terms of cents, so should be an int.

So far, the CheckingAccount class looks like this:

class CheckingAccount

{

// instance variables

String accountNumber;

String accountHolder;

int balance;

*constructors*

*methods*

}

Next, consider the constructor. The constructor has the same name as the class, and looks like this:

CheckingAccount( *parameter-list* )

{

*initialize the data*

}

The constructor is used with the new operator to create a new checking account object. It then initializes the account number, the account holder's name, and starting balance. Actual parameters are supplied to the constructor when it is used, such as in the following:

CheckingAccount billsAccount =

new CheckingAccount( "123", "Bob", 100 ) ;

This statement creates a new CheckingAccount object by calling the constructor. The constructor will initialize the object's accountNumber to "123", its accountHolder to "Bob", and its balance to 100.

**QUESTION 6:**

Fill in the parameter list for the constructor. You will have to think of names for the parameters, and will have to include the type of each parameter.

Top of Form

CheckingAccount(   ,   ,  )

{

*initialize the data*

}

### Answer:

CheckingAccount( String accNumber, String holder, int start )

{

initialization of data

}

Since parameters are used only inside the constructor, they often have shorter, less descriptive names than the names given to more permanent data.

# Completing the Constructor

So far, the class looks like this:

Top of Form

class CheckingAccount

{

// instance variables

String accountNumber;

String accountHolder;

int balance;

//constructors

CheckingAccount( String accNumber, String holder, int start )

{

 =  ;

 =  ;

 =  ;

}

methods

}

Bottom of Form

Next, complete the constructor with three statements that initialize the object's data.

### QUESTION 7:

Complete the constructor.

### Answer:

//constructors

CheckingAccount( String accNumber, String holder, int start )

{

accountNumber = accNumber ;

accountHolder = holder ;

balance = start ;

}

# Test Program

We have enough code to put together a test program. The test program will not do much, but it will compile and run.

class CheckingAccount

{

// instance variables

String accountNumber;

String accountHolder;

int balance;

//constructors

CheckingAccount( String accNumber, String holder, int start )

{

accountNumber = accNumber ;

accountHolder = holder ;

balance = start ;

}

// methods

}

class CheckingAccountTester

{

public static void main( String[] args )

{

CheckingAccount account1

= new CheckingAccount( "123", "Bob", 100 );

System.out.println( account1.accountNumber + " " +

account1.accountHolder + " " + account1.balance );

}

}

This program can be copied to a file, compiled, and run in the usual way. The output will be:

C:\chap32>java CheckingAccountTester

123 Bob 100

### QUESTION 8:

What does the expression   account1.accountNumber   mean? (Look in the println statement to see where this expression was used.)

### Answer:

The expression means to use the variable accountNumber that is part of the object referred to by account1.

# What Members of an Object can be Seen?

The **dot operator** is how you access a member of an object:

referenceToAnObject **.** partOfTheObject

Of course, an object has to exist, and there has to be a reference to it. In the program, this happened when the object was constructed:

CheckingAccount account1

= new CheckingAccount( "123", "Bob", 100 );

When a method has a reference to an object, the method can:

1. access the object's instance variables using dot notation, and
2. call the object's methods using dot notation.

However, it is common for a class to have **private** variables and methods. Only methods of the class can access these. This will be discussed further in the next chapter.

### QUESTION 9:

Can the main() method of TesterClass see the statements in the methods of SomeClass?

**Answer:**

No—this is not allowed and the dot notation does not even give you a way to ask for it.

**Method Definition**

Now that we have a test program, we can add methods to the class one-by-one, testing each method as it is added. Recall the three methods from the requirements:

* Accept a deposit
* Process a check
* Get the current balance

Remember the syntax for method definition:

*returnType* *methodName* ( *parameterList* )

{

*statementList*

}

The first line in the above is called the **signature** of a method definition. It does not have to be all on one line. The returnType is the data type that the method returns. It will be one of the primitive data types, or a class. The methodName is an identifier picked by the programmer. It can be any identifier except for reserved words or identifiers already in use. The parameterList is a list of parameters and their data types. If there are no parameters, the parameter list is omitted, (but the two parentheses must be there).

**QUESTION 10:**

(Review: ) What is *returnType* when a method does not return a value?

**Answer:**

void

**A Method to Get the Balance**

Here is the class definition, again:

class CheckingAccount

{

// instance variables

String accountNumber;

String accountHolder;

int balance;

//constructors

. . . .

// methods

}

Let us work on a method that will return the current balance. When it is called, it will not alter any data in a checking account object, but will merely return the current balance.

**QUESTION 11:**

Write the first line of the "current balance" method. The method will not use any parameters. You will have to think of a name for the method.

### Answer:

int getBalance()

Other names for the method would also work, of course.

# Implementing the Method

The method returns the current balance, which is an int so the return type is int. The method does not need any input data, so its parameter list is empty. The two parentheses () are needed, even though there is nothing inside of them. So far, the class looks like this:

Top of Form

class CheckingAccount

{

// instance variables

String accountNumber;

String accountHolder;

int balance;

//constructors

. . . .

// methods

int getBalance()

{

return  ;

}

}

Bottom of Form

All getBalance() needs is a statement that returns the balance of the checking account.

### QUESTION 12:

Complete the method by filling in the blank.

### Answer:

class CheckingAccount

{

// methods

int getBalance()

{

return balance;

}

}

# Testing the Method

When the getBalance() method is added, the program can be run and tested. It is a good idea to alternate between writing and testing like this. By testing each part as it is added you catch problems early on, before they can cause further problems. Of course, it helps to have a good initial design. This is how houses are built, after all. You start with a good design, then lay the foundation. Then structures are built and tested until the house is complete.

Here is a compilable and runable test program:

class CheckingAccount

{

// instance variables

String accountNumber;

String accountHolder;

int balance;

//constructors

CheckingAccount( String accNumber, String holder, int start )

{

accountNumber = accNumber ;

accountHolder = holder ;

balance = start ;

}

// methods

int getBalance()

{

return balance;

}

}

class CheckingAccountTester

{

public static void main( String[] args )

{

CheckingAccount account1 =

new CheckingAccount( "123", "Bob", 100 );

CheckingAccount account2 =

new CheckingAccount( "92a-44-33", "Kathy Emerson", 0 );

System.out.println( account1.accountNumber + " " +

account1.accountHolder + " " + account1.getBalance() );

System.out.println( account2.accountNumber + " " +

account2.accountHolder + " " + account2.getBalance() );

}

}

This test program creates two objects, account1 and account2 of type CheckingAccount.

### QUESTION 13:

What will the program output when run? (Sketch out the answer, don't worry about spaces.)

### Answer:

123 Bob 100

92a-44-33 Kathy Emerson 0

# Method to Accept a Deposit

Next, implement the method that accepts a deposit. Deposits will be expressed in cents. The method will have one parameter, the number of cents to be added to the balance. The method will not return a value, so its return type will be void.

class CheckingAccount

{

// instance variables

String accountNumber;

String accountHolder;

int balance;

//constructors

. . . .

// methods

. . . .

void  ( int  )

{

balance =  +  ;

}

}

Since the return type of the method is void, no return statement is needed. When the method is run, it will automatically return to the place where it was called from after the last statement of the method is executed.

### QUESTION 14:

Complete the method by filling in the blanks.

### Answer:

class CheckingAccount

{

// instance variables

String accountNumber;

String accountHolder;

int balance;

//constructors

. . . .

// methods

. . . .

void processDeposit ( int amount )

{

balance = balance + amount ;

}

}

# Method to Process Checks

If you have the previous test program in a file, it would be nice to add the processDeposit() method and test it. The method to process a check is slightly more complicated:

* Assume that the amount of a check is a positive value, expressed in cents.
* If the current balance is less than $1000.00, there is a $0.15 processing charge.
* The amount of the check and the processing charge (if any) are subtracted from the balance.
* The return type is void.

Here is a sketch of the method:

Top of Form

void processCheck( int  )

{

int charge;

if (  < 100000 )

charge =  ;

else

charge =  ;

balance =  -  -  ;

}

Bottom of Form

### QUESTION 15:

Fill in the blanks to complete the method.

### Answer:

void processCheck( int amount )

{

int charge;

if ( balance < 100000 )

charge = 15;

else

charge = 0;

balance = balance - amount - charge ;

}

# Complete Class

The complete class is defined as:

class CheckingAccount

{

// instance variables

String accountNumber;

String accountHolder;

int balance;

//constructors

CheckingAccount( String accNumber, String holder, int start )

{

accountNumber = accNumber ;

accountHolder = holder ;

balance = start ;

}

// methods

int getBalance()

{

return balance ;

}

void processDeposit( int amount )

{

balance = balance + amount ;

}

void processCheck( int amount )

{

int charge;

if ( balance < 100000 )

charge = 15;

else

charge = 0;

balance = balance - amount - charge ;

}

}

### QUESTION 16:

Now that the coding is complete, are we done?

### Answer:

Of course not. Now there is all the fun of testing the implementation.

# More Testing

Here is an expanded test program for the class. The class itself is given in the previous page, so if you are going to run this program (could there be any doubt?) you will have to copy and paste two things into your text editor.

class CheckingAccount

{

. . . .

}

class CheckingAccountTester

{

public static void main( String[] args )

{

CheckingAccount account1 = new CheckingAccount( "123", "Bob", 100 );

System.out.println( account1.getBalance() );

account1.processDeposit( 2000 );

account1.processCheck( 1500 );

System.out.println( account1.getBalance() );

}

}

Another way to organize this program is to copy the CheckingAccount class into a file CheckingAccount.java. Then copy the CheckingAccountTester class on this page into a file CheckingAccountTester.java in the same disk directory. Then enter the commands

C:\chap32\> javac CheckingAccount.java CheckingAccountTester.java

C:\chap32\> java CheckingAccountTester

The name of the file for the java command is the one that contains main() .

### QUESTION 17:

What does the output of this test program look like?

### Answer:

100

585

(Remember to subtract the 15 cent service charge.)

# Scaffolding

If this class were to be used in an actual bank, it would need a great deal more testing. Usually testing involves placing statements that write out information to the terminal as the program is being run. Thoughtful placement of these statements can greatly ease software development. These statements (and other statements intended to be used for testing and program development) are sometimes called **scaffolding**. They are put in place as the program is being written and tested, and are removed when testing is finished. This is similar to the scaffolding used when a building is being constructed. Well-placed scaffolding greatly aids the carpenters, roofers, painters and other trades that work on the building. Programmers should do the same for themselves.

Top of Form

class CheckingAccount

{

// instance variables

String accountNumber;

String accountHolder;

int balance;

//constructors

. . . .

// methods

. . . .

void display()

{

System.out.println(  );

}

}

Bottom of Form

The statements that write out the state of a CheckingAccount object are somewhat awkward:

System.out.println( account1.accountNumber + " " +

account1.accountHolder + " " + account1.getBalance() );

It would aid in testing if a CheckingAccount itself could be asked to do this. This is a display method which can be added to the class, as seen above.

### QUESTION 18:

Write a complete display() method.

### Answer:

void display()

{

System.out.println( accountNumber + "\t" + accountHolder + "\t" + balance );

}

(The string "\t" tells the compiler that you want the tab character.)

# Using the display() Method

Since this method is a member of a CheckingAccount object, the object's data is accessed by using the variable name, like accountNumber. The dot operator is needed outside of the object, like account1.accountNumber in the main method.

Another nice thing to do is to use the tabulation character "\t" to align the output better. When the display() has been defined, the testing program can be more easily written:

class CheckingAccount

{

. . . . (Now including the display() method.)

}

class CheckingAccountTester

{

public static void main( String[] args )

{

CheckingAccount account1 = new CheckingAccount( "123", "Bob", 100 );

account1.display() ;

account1.processDeposit( 2000 );

account1.processCheck( 1500 );

account1.display() ;

}

}

### QUESTION 19:

With this nice, new scaffolding in place, you must surely be eager to do further testing of the program. Add statements after the old statements in the test program that:

* Create a new object, account2
  + The account number is "007"
  + The account holder is Bond. James Bond.
  + The account amount is $500 (The end of the cold war has been hard on Mr. Bond.)
* Display account2.
* Deposit $700 in account2.
* Process a $100 check.
* Display account2.

### Answer:

Sure. A checking account has identity, state, and behavior, just like a software object.

# Establishing the Requirements

This chapter develops a checking account class. To start, decide what checking account objects do and what data they hold. This process is called **requirements analysis** and is the most important phase of writing big programs.

The companies of the late, lamented, dot-com bubble were strong on sock-puppets and weak on requirements analysis.

### QUESTION 2:

Think of three data items that will be part of a checking account.

### Answer:

public static void main( String[] args )

{

CheckingAccount account1 = new CheckingAccount( "123", "Bob", 100 );

account1.display() ;

account1.processDeposit( 2000 );

account1.processCheck( 1500 );

account1.display() ;

CheckingAccount account2 = new CheckingAccount( "007", "James Bond", 50000 );

account2.display() ;

account2.processDeposit( 70000 );

account2.processCheck( 10000 );

account2.display() ;

}

# Further Testing

When the program is run, you will see:

123 Bob 100

123 Bob 585

007 James Bond 50000

007 James Bond 110000

### QUESTION 20:

Now try to do something a little bit tricky. Say that James wrote out a $300 check to Bob, and that Bob deposited the check in Bob's account. Add statements (after all the others in the mainmethod) that do this.

### Answer:

public static void main( String[] args )

{

. . . . . .

int check = 30000;

account2.processCheck( check );

account1.processDeposit( check );

account1.display();

account2.display();

}

# Aliasing (Review)

This is not really part of testing this class, but it is convenient to mention aliasing again. Recall that there can be more than one reference to a given object. Each reference is called an **alias**. Here is another test program, set up to show this:

class CheckingAccount

{

. . . .

}

class CheckingAccountTester

{

public static void main( String[] args )

{

CheckingAccount account1 = new CheckingAccount( "123", "Bob", 100 );

CheckingAccount account2 = new CheckingAccount( "456", "Jill", 900 );

CheckingAccount account3;

account1.display() ;

account2.display() ;

account3 = account1;

account3.display() ;

}

}

### QUESTION 21:

* How many CheckingAccount objects are there?
* How many object reference variables are there?
* What will the program print out?

### Answer:

* How many CheckingAccount objects are there?   Two
* How many object reference variables are there?   Three
* What will the program print out?

123 Bob 100

456 Jill 900

123 Bob 100

# Review of ==

The == (equal-equal) operator is an alias detector. It checks if two reference variables refer to the same object. It does not actually look at the objects. The following program segment illustrates this:

{

CheckingAccount account1 = new CheckingAccount( "123", "Bob", 100 );

CheckingAccount account2 = new CheckingAccount( "456", "Jill", 900 );

CheckingAccount account3;

account3 = account1;

if ( account1 == account 3 )

System.out.println("An alias has been detected!");

else

System.out.println("These are different objects!");

}

It will print out An alias has been detected!.

### QUESTION 22:

Say that the following lines are added to the program (immediately following the lines already there):

account3.processCheck( 85 ); // subtract 100 cents, including service charge

account1.display();

account3.display();

What will be printed?

### Answer:

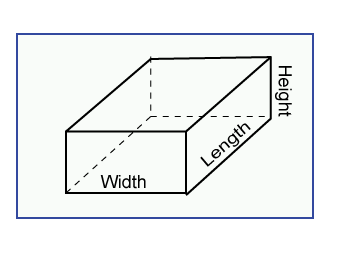
An alias has been detected!

123 Bob 0

123 Bob 0

The single object referred to by the aliases account1 and by account3 was changed by the method call account3.processCheck( 85 ).

**Fill in the Blanks**



1.       **Design of the Box Class**.     Objects of this class will represent a *box* (such as a cardboard box.) The sides of a box are rectangles. A *Box* will have three essential characteristics: width, height, and length. From these characteristics other values may be calculated: volume and total surface area.

Fill in the blanks in the following design for the class:

Top of Form

**class** Box

A class that models a cardboard box.

**Constructors**

Box ( double Width,

double Length,

double Height )

*The constructor should not include volume and area as parameters because these values can be calculated from the width, height, and length of the box. As a convenience, include a constructor to use when all sides of the box are the same size (when it is a cube):*

Box ( double )

**Methods**

// calculate the volume of the box

double volume()'">

// calculate the total surface area of the box

double area()'">

Bottom of Form

2.      **Checking the Design**.    To check the design, write a small program that uses the class. Write a program that creates a Box object with sides of 2.5, 5.0, and 6.0 inches. Write out its surface area and volume.

Top of Form

class BoxTester

{

public static void main ( String[] args )

{

Box box = new ;

System.out.println( "Area: " + box.area() +

" volume: " + box.Volume() );

}

}

Bottom of Form

3.      **Skeleton of the Class**.    Fill in the blanks that give the over-all design of the class.

Top of Form

class Box

{

//

//

//

}

Bottom of Form

4.      **Fill in Instance Variables**.    Fill in the data type of each instance variable.

Top of Form

class Box

{

// Instance Variables

length ;

width ;

height ;

// Constructors

// Methods

}

Bottom of Form

The variables should be floating point because integer dimensions would be too much of a restriction. Nearly always, if you use floating point you should use double.

5.      **Complete the Constructors**.    Constructors initialize the instance variables of the objects being constructed.

Top of Form

class Box

{

// Instance Variables

double length ;

double width ;

double height ;

// Constructors

( double , double , double )

{

this.width = ;

this.height = ;

this.length = ;

}

// initialize all sides to the same value

( double )

{

width = ;

height = ;

length = ;

}

// Methods

}

Bottom of Form

When an instance variable and a parameter use the same identifier, you specify the instance variable of the object by saying this.identifier as in the above.

6.      **Complete a Method**.     The specification for the volume[[egg]] method says it looks like this:

// calculate the volume of the box

double volume()

The formula to use is: volume = product of all three sides

Top of Form

class Box

{

// Instance Variables

double length ;

double width ;

double height ;

// Constructors

Box ( double width, double height, double length )

{

this.width = width ;

this.height = height ;

this.length = length ;

}

Box ( double side )

{

width = side ;

height = side ;

length = side ;

}

// Methods

double volume()

{

return \* \* ;

}

}

Bottom of Form

(Of course, the three instance variables could be used in any order in the arithmetic expression.)

7.      **Complete the other Method**.     The specification for the area() method says it looks like this: double area() . There are three pairs of sides to a box. Opposite sides have the same area, so the calculation looks like this: 2 \* (sum of area of each unique side)

Top of Form

class Box

{

// Instance Variables

double length ;

double width ;

double height ;

// Constructors

Box ( double width, double height, double length )

{

this.width = width ;

this.height = height ;

this.length = length ;

}

Box ( double side )

{

width = side ;

height = side ;

length = side ;

}

// Methods

double volume()

{

return width \* height \* length ;

}

double area()

{

return 2 \* ( \* +

\* +

\* ) ;

}

}

Bottom of Form

8.      **Testing Class**.     At this point all the coding for the definition of Box is done. Say that the small test program (class BoxTester) is in a file called BoxTester.java and that the complete definition of the Box class is in a file called Box.java.

Top of Form

class BoxTester

{

public static void main ( String[] args )

{

// create a box with sides= 2.5, 3.0, and 5.0

Box box1 = new Box( 2.5, 3.0, 5.0 );

System.out.println( "Box1 Area: " + box1.area() +

" Volume: " + box1.volume() );

// create a box with all sides = 3.0

Box box2 = Box ( );

System.out.println( "Box2 Area: " + box2.area() +

" Volume: " + box2.volume() );

}

}

Bottom of Form

9.      **Compile each File**.     Each of the two files must be compiled:

Top of Form

C:\MyFiles>javac

C:\MyFiles>javac

Bottom of Form

10.      **Run the Program**.     The java bytecode interpreter must be started with the \*.class file that contains the *main()* method:

Top of Form

C:\MyFiles>java

Bottom of Form

End of the Exercise. If you want to do it again, click on "Refresh" in your browser window.

# Encapsulation

So far, the objects that we have designed have instance variables that are visible outside of the objects. Code that holds a reference to an object can change the object's instance variables. This can lead to problems.

A better design enforces **encapsulation**. Some (or all) of an object's instance variables are visible only to the object's own methods. Code outside of the object cannot access these variables directly. This chapter discusses how to use the private visibility modifier to do this.

#### Chapter Topics:

* The private and the public visibility modifiers
* Default visibility
* Encapsulation
* Access methods
* Enhancement of the CheckingAccount class

### QUESTION 1:

Should a bank control who has access to your checking account?

### Answer:

Yes.

# The private Visibility Modifier

When a member of a class is declared private it can be used only by the methods of that class. Here is the checking account class definition from the last chapter with each of its variables declared to be private.

class CheckingAccount

{

// variable declarations

private String accountNumber;

private String accountHolder;

private int balance;

//constructor

CheckingAccount( String accNumber, String holder, int start )

{

accountNumber = accNumber ;

accountHolder = holder ;

balance = start ;

}

// methods

int getBalance()

{

return balance ;

}

void processDeposit( int amount )

{

balance = balance + amount ;

}

void processCheck( int amount )

{

int charge;

if ( balance < 100000 )

charge = 15;

else

charge = 0;

balance = balance - amount - charge ;

}

}

Now only the methods of a CheckingAccount object can "see" accountNumber, accountHolder, and balance.

### QUESTION 2:

It is useless to have a checking account if the balance cannot be changed. How can you change the balance in a CheckingAccount object?

### Answer:

Use processDeposit() or processCheck().

# Access Methods

A class with private data provides access to that data through **access methods**. An access method is a method which uses the private data of its object and is visible to other classes. Some access methods alter data; others return a value but don't alter data. Here is a main() that uses the access methods of a CheckingAccount object.

class CheckingAccount

{

private String accountNumber;

private String accountHolder;

private int balance;

. . . .

}

class CheckingAccountTester

{

public static void main( String[] args )

{

CheckingAccount bobsAccount = new CheckingAccount( "999", "Bob", 100 );

System.out.println( bobsAccount.getBalance() );

bobsAccount.processDeposit( 200 );

System.out.println( bobsAccount.getBalance() );

}

}

This main() program uses access methods of the CheckingAccount object to change the object's data. This is (usually) the correct way to use an object. The methods of an object know best how to change its instance variables. The idea of private is to **enforce** this correct use.

### QUESTION 3:

Could an access method check for errors and change instance variables only if the data is correct?

### Answer:

Yes.

# main() Can't See Private Data

The main() method changes the CheckingAccount object by using the object's access methods. Here is a different main() that does not use the access methods.

class CheckingAccount

{

private String accountNumber;

private String accountHolder;

private int balance;

. . . .

}

class CheckingAccountTester

{

public static void main( String[] args )

{

CheckingAccount bobsAccount = new CheckingAccount( "999", "Bob", 100 );

System.out.println( bobsAccount.balance );

bobsAccount.balance = bobsAccount.balance + 200;

System.out.println( bobsAccount.balance );

}

}

### QUESTION 4:

Is there a problem with this program?

### Answer:

Yes. You have tried to access private data from "outside" the object. The compiler lets you know what it thinks about that:

compiling: CheckingAccountTester.java

CheckingAccountTester.java:46:

Variable balance in class CheckingAccount not accessible from class CheckingAccountTester.

System.out.println( bobsAccount.balance );

^

CheckingAccountTester.java:47:

Variable balance in class CheckingAccount not accessible from class CheckingAccountTester.

bobsAccount.balance = bobsAccount.balance + 200;

^

CheckingAccountTester.java:47:

Variable balance in class CheckingAccount not accessible from class CheckingAccountTester.

bobsAccount.balance = bobsAccount.balance + 200;

^

CheckingAccountTester.java:48:

Variable balance in class CheckingAccount not accessible from class CheckingAccountTester.

System.out.println( bobsAccount.balance );

^

4 errors

# Careful Access Control

It may seem a bit silly that the CheckingAccount class uses private to prevent main() from seeing its variables, but then provides methods so that main() can access them anyway. The idea of this is that the access methods have control over each access to the private data. For example, a programmer can't increase the balance of a checking account by writing:

bobsAccount.balance = bobsAccount.balance + 200;

To increase the balance, the processDeposit() method must be used. A more elaborate method might check that it was OK to proceed before adding the deposit to the balance. It might check that the account has not been closed, might ask for a password before it allows access, and might log every change in a history file.

When data is private the only changes to it are made through a small number of access methods. This helps keep objects consistent and bug-free. If a bug is detected, there are only a few places to look for it.

### QUESTION 5:

(Test of your memory: ) In the checking account example, what is the minimum balance before the 15 cents charged per check is dropped?

### Answer:

I don't remember that.

# Consistent Application of a Rule

Actually, the amount is $1000. Luckily, that "rule" is programmed into to the processCheck() access method. By using the access method, the rule is correctly applied, regardless of your poor memory:

bobsAccount.processCheck( 1400 ); // correctly applies the rule

jillsAccount.processCheck( 4500 ); // correctly applies the rule

If there is a bug in the application of the rule, there is only one place to look for the problem: in the access method.

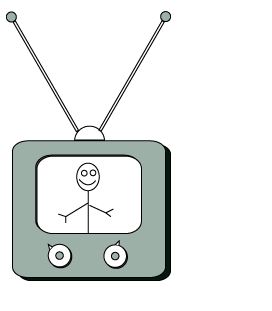
### QUESTION 6:

Another bank charges 10 cents per check for accounts with a balance of more that $500. Our bank needs to do the same in order to keep its customers. How hard will it be to make that change?

**Answer:**

Not hard at all. Just one method needs to be changed — the access method that implements the rule. If balance was not marked private you would have to look over every line of the entire program to find places that changed it.

**Encapsulation**



The programming technique we have been talking about is called *encapsulation*.

**Encapsulation** means hiding the details of an object from the other parts of a program. The object can be used only through its access methods, which are carefully written to keep the object consistent and secure.

Encapsulation makes an object look like a **black box**: The insides of the box are hidden from view. Controls are on the outside of the box. The user can change the operation of the box only by using the controls.

The usual example of this is a TV set where the inner workings are sealed in a box. The user interacts with the set using some well-defined controls. The controls are sometimes called the **user interface**. In object oriented programming, programmers try to make the interface to the object simple and useful. The inner workings of the object should be made private.

**QUESTION 7:**

(Thought question: ) Do you think that it is possible to have a private method?

**Answer:**

Yes. The idea of encapsulation is to hide the details of an object from other sections of the software. Some of the details might be methods.

**Private Methods**

A **private method** of an object can be used only by the other methods of the object. Parts of a program outside of the object cannot directly use a private method of the object.

Say that the bank wants to keep track of how many times each the balance of checking account has been altered. (This might be done as a security measure.) To do this, a **use count** is added to the data of the CheckingAccount class.

class CheckingAccount

{

// data-declarations

private String accountNumber;

private String accountHolder;

private int balance;

private int useCount = 0;

private void incrementUse()

{



}

void processDeposit( int amount )

{

incrementUse();

balance = balance + amount ;

}

void processCheck( int amount )

{

int charge;

incrementUse();

if ( balance < 100000 )

charge = 15;

else

charge = 0;

balance = balance - amount - charge ;

}

// other methods

. . .

}

The processDeposit() and processCheck() methods call incrementUse() to increment the use count each time they are used. We want the use count to change for these two reasons only, so the incrementUse() method and the variable useCount are made private.

**QUESTION 8:**

Fill in the blank so that the new private method increments the use count.

### Answer:

private void incrementUse()

{

useCount = useCount + 1;

}

# main() Can't use a Private Method

Here is a main() that mistakenly tries to change useCount:

class CheckingAccount

{

private String accountNumber;

private String accountHolder;

private int balance;

private int useCount = 0;

. . . .

private void incrementUse()

. . . .

}

class CheckingAccountTester

{

public static void main( String[] args )

{

CheckingAccount bobsAccount = new CheckingAccount( "999", "Bob", 100 );

bobsAccount.processCheck( 50 );

bobsAccount.incrementUse();

bobsAccount.useCount = 15;

}

}

### QUESTION 9:

Will this program compile and execute?

### Answer:

No. The program will not compile.

# Compiler Complaints

Here is what the compiler outputs for the mistaken program:

compiling: CheckingAccountTester.java

CheckingAccountTester.java:55:

No method matching incrementUse() found in class CheckingAccount.

bobsAccount.incrementUse();

^

CheckingAccountTester.java:56:

Variable useCount in class CheckingAccount not accessible from class CheckingAccountTester.

bobsAccount.useCount = 15;

^

2 errors

The main() method cannot use the private variable useCount nor can it use the private method incrementUse(). These can only be used by the other methods of the object.

The main() method can use bobsAccount.processCheck() which is not private. It in turn uses the private method incrementUse(). This is OK.

### QUESTION 10:

Does using private increase the security of Java programs for the Web?

### Answer:

Yes. Java is used for Web programming because of its convenient security features.

# The public Visibility Modifier

The private visibility modifier keeps outsiders from looking in. However, the access methods are intended for outsiders, and must be visible to outsiders in order to be useful. The **public** access modifier explicitly says that a method or variable of an object can be accessed by code outside of the object.

The public visibility modifier is usually used for all access methods and constructors in a class definition. Most variables are made private. Here is a skeleton of the CheckingAccount class:

Top of Form

class CheckingAccount

{

private String accountNumber;

private String accountHolder;

private int balance;

private int useCount = 0;

CheckingAccount( String accNumber, String holder, int start ) { . . . . }

void incrementUse() { . . . . }

int getBalance() { . . . . }

void processDeposit( int amount ) { . . . . }

void processCheck( int amount ) { . . . . }

}

Bottom of Form

### QUESTION 11:

Pick a visibility modifier for the constructor and for each of the methods.

### Answer:

class CheckingAccount

{

private String accountNumber;

private String accountHolder;

private int balance;

private int useCount = 0;

public CheckingAccount( String accNumber, String holder, int start ) { . . . . }

private void incrementUse() { . . . . }

public int getBalance() { . . . . }

public void processDeposit( int amount ) { . . . . }

public void processCheck( int amount ) { . . . . }

}

# toString() Method

It would be nice to have a toString() method in this class that shows the use count as well as the other data. There is already a toString() method. All classes automatically have such a method. (This is done by inheritance, and subject of a upcoming chapter.) But the automatically supplied method may not do what you want.

If you write your own toString() method it will replace the automatically supplied one. The method must look like this:

public String toString()

{

}

The method must be declared to be public

class CheckingAccount

{

private String accountNumber;

private String accountHolder;

private int balance;

private int useCount = 0;

. . . .

public String toString()

{

return "Account: " + accountNumber + "\tName: " + accountHolder +

"\tBalance: " + balance + "\tUse Count: " +  ;

}

}

### QUESTION 12:

Modify the method so that it also prints out the use count.

### Answer:

public String toString()

{

return "Account: " + accountNumber + "\tName: " + accountHolder +

"\tBalance: " + balance + "\tUse Count: " + useCount;

}

# Complete Class

Here is the complete class:

class CheckingAccount

{

// data-declarations

private String accountNumber;

private String accountHolder;

private int balance;

private int useCount = 0;

//constructors

CheckingAccount( String accNumber, String holder, int start )

{

accountNumber = accNumber ;

accountHolder = holder ;

balance = start ;

}

// methods

private void incrementUse()

{

useCount = useCount + 1;

}

int getBalance()

{

return balance ;

}

void processDeposit( int amount )

{

incrementUse();

balance = balance + amount ;

}

void processCheck( int amount )

{

int charge;

incrementUse();

if ( balance < 100000 )

charge = 15;

else

charge = 0;

balance = balance - amount - charge ;

}

public String toString()

{

return "Account: " + accountNumber + "\tName: " + accountHolder +

"\tBalance: " + balance + "\tUse Count: " + useCount;

}

}

### QUESTION 13:

What does the useCount keep track of:

* The number of times an individual object has been used, or
* The number of times the entire class has been used?

### Answer:

The number of times each individual object has been used. (If this is not clear, look at the class definition again to see that the variable useCount is part of each object, and that each object has its own method incrementUse() which increments its own variable.)

# Example main()

Here is a main() that shows these ideas:

class CheckingAccountTester

{

public static void main( String[] args )

{

CheckingAccount bobsAccount = new CheckingAccount( "999", "Bob", 100 );

CheckingAccount jillsAccount = new CheckingAccount( "111", "Jill", 500 );

bobsAccount.processCheck( 50 );

System.out.println( bobsAccount.toString() );

jillsAccount.processDeposit( 500 );

jillsAccount.processCheck( 100 );

jillsAccount.processCheck( 100 );

jillsAccount.processDeposit( 100 );

System.out.println( jillsAccount.toString() );

}

}

### QUESTION 14:

How may times has Bob's account been used?

How may times has Jill's account been used?

### Answer:

The use count is in the program's output:

Account: 999 Name: Bob Balance: 35 Use Count: 1

Account: 111 Name: Jill Balance: 270 Use Count: 4

# Default Visibility

If you do not specify public or private for a variable or a method, then it will have default visibility. Default visibility allows a variable or method to be seen by all methods of a class or other classes that are part of the same **package**. A package is a group of related classes.

For now, default visibility means about the same thing as public visibility. But it is best to explicitly declare members public if that is what you actually need. You can not explicitly declare a member of a class to have default visibility (for example, you can't say default int monthlyCharge; )

Later on, after these notes have discussed inheritance there will be other visibility modifiers, and additional rules for public and private visibility.

### QUESTION 15:

Should a constructor be made public or private?

### Answer:

Almost always a constructor will have public visibility so that objects of the class can be constructed by "outsiders" such as the main() of a testing class.

**Instructions:** For each question, choose the single best answer. Make your choice by clicking on its button. You can change your answers at any time. When the quiz is graded, the correct answers will appear in the box after each question.

Top of Form

1. Why is the *main()* method special in a Java program?

 **A.** It is where the Java interpreter starts the whole program running.

 **B.** Only the *main()* method may create objects.

 **C.** Every class must have a *main()* method.

 **D.** The *main()* method must be the only *static* method in a program.



Bottom of Form

Top of Form

2. Which of the following is the general scheme for a class definition:

 **A.**

Class ClassName

{

// Description of the instance variables.

// Description of the constructors.

// Description of the methods.

}

 **B.**

class ClassName

{

// Description of the instance variables.

// Description of the constructors.

// Description of the methods.

}

 **C.**

ClassName

{

// Description of the instance variables.

// Description of the constructors.

// Description of the methods.

}

 **D.**

class ClassName

{

public static void main ( String[] args )

{

// entire program goes here

}

}



Bottom of Form

Top of Form

3. Here is the general syntax for *method* definition:

accessModifier returnType methodName( parameterList )

{

Java statements

return returnValue;

}

What is true for the *accessModifier*?

 **A.** It must always be *private* or *public*.

 **B.** It can be omitted, but if not omitted it must be *private* or *public*.

 **C.** It can be omitted, but if not omitted there are several choices, including *private* and *public* .

 **D.** The *access modifier* must agree with the type of the *return value*.



Bottom of Form

Top of Form

4. When the *access modifier* is omitted from the definition of a member of a class (instance variable or method) the member has ..... ?

 **A.** default access.

 **B.** public access.

 **C.** private access.

 **D.** universal access.



Bottom of Form

Top of Form

5. Here is the general syntax for *method* definition:

accessModifier returnType methodName( parameterList )

{

Java statements

return returnValue;

}

What is true for the *returnType* and the *returnValue?*

 **A.** The *returnValue* must be exactly the same type as the *returnType.*

 **B.** The *returnValue* must be the same type as the *returnType,* or be of a type that can be converted to *returnType* without loss of information.

**c.** The *returnValue* can be any type, but will be automatically converted to *returnType* when the method returns to the caller.

**d.** If the *returnType* is *void* then the *returnValue* can be any type.



Bottom of Form

Top of Form

6. What term is used for hiding the details of an object from the other parts of a program?

 **A.** Obfustication.

 **B.** Data Mining.

 **C.** Compilation.

 **D.** Encapsulation.



Bottom of Form

Top of Form

7. What is the effect of giving a class member *private* access?

 **A.** When a member of a class is declared *private* it can be used in only one place in a program.

 **B.** When a member of a class is declared *private* it can be used only in methods that are members of that class.

 **C.** When a member of a class is declared *private* it can only be used by other *private* members of other classes.

 **D.** When a member of a class is declared *private* there will be only one instance of it, no matter how many objects are instantiated.



Bottom of Form

Top of Form

8. Methods of a class that are used by "outsiders" to access *private* (and other) data of the class are called...

 **A.** Access methods.

 **B.** Private methods.

 **C.** Public methods.

 **D.** Member methods.



Bottom of Form

Top of Form

9. What will happen if a *main()* method of a "testing" class tries to access a *private* instance variable of an object using dot notation?

 **A.** The compiler will find the error and will not make a .class file.

 **B.** The compiler will automatically change the *private* variable to a *public* variable.

 **C.** The program will compile successfully, but the .class file will not run correctly.

 **D.** The program will compile and run successfully.



# Parameters, Local Variables,  and Overloading

The state of an object consists of the data it holds in its instance variables. Instance variables hold their values until they are explicitly changed or until the object is destroyed.

An object's methods frequently work with other values that are not held in instance variables. These values are held in **local variables** and **parameters**. This chapter discusses how these are declared and used.

#### Chapter Topics:

* Formal and Actual Parameters
* Scope of Parameters
* Local Variables
* Scope of Local Variables
* Method Overloading
* Signature of a Method

### QUESTION 1:

(Reviewsmile What is a parameter of a method?

### Answer:

A parameter is a value that is sent to a method when it is called.

# Parameters

Here is an example of a method that uses a parameter.

class CheckingAccount

{

. . . .

private int balance;

. . . .

public void processDeposit( int amount )

{

balance = balance + amount ;

}

}

The parameter amount is used by a caller to send a value to the method. This is called **passing a value** into the method. Here is part of a main() method that uses the parameter to pass a value into the processDeposit() method of an object:

class CheckingAccountTester

{

public static void main( String[] args )

{

CheckingAccount bobsAccount = new CheckingAccount( "999", "Bob", 100 );

bobsAccount.processDeposit( 200 );

. . . . . .

}

}

When the statement

bobsAccount.processDeposit( 200 );

is executed, the parameter amount is given the value 200. Now, as the processDeposit method executes, that value is added to the object's instance variable balance:

balance = balance + amount ;

Then the method finishes and control returns to main(). The balance of the bobsAccount object has been changed.

### QUESTION 2:

1. Will the instance variable balance of the object hold its value permanently?
2. Will the parameter amount of the object's method hold its value permanently?

### Answer:

1. Will the instance variable balance hold its value permanently?
   * Yes — balance is part of the state of the object and holds its value as long as the object exists.
2. Will the parameter amount hold its value permanently?
   * No — amount is used only to pass in a value into the method. It is not permanent.

# Formal and Actual Parameters

An *identifier* is a name used for a class, a variable, a method, or a parameter. The following definitions are useful:

* **formal parameter** — the identifier used in a method to stand for the value that is passed into the method by a caller.
  + For example, amount is a formal parameter of processDeposit
* **actual parameter** — the actual value that is passed into the method by a caller.
  + For example, the 200 used when processDeposit is called is an actual parameter.

When a method is called, the formal parameter is temporarily "bound" to the actual parameter. The method uses the formal parameter to stand for the actual value that the caller wants to be used.

For example, here the processDeposit method uses the formal parameter amount to stand for the actual value used in the procedure call:

balance = balance + amount ;

**Note:** formal parameters are bound to an actual value only as long as their method is active. When a method returns to its caller, the formal parameters no longer contain any values. They cannot be used to store the state of an object.

### QUESTION 3:

What is used to store the state of an object?

### Answer:

Instance variables are used to store the state of an object. They hold values for as long as the object exists.

# Parameters are Seen by their Own Method, Only

The formal parameters of a method can be "seen" only by the statements of their own method. This means that if a method tries to use a parameter of some other method, the compiler will find a syntax error.

Here is the CheckingAccount class again, this time with a new definition of the toString() method

class CheckingAccount

{

. . . .

private int balance;

. . . .

public void processDeposit( int amount )

{

System.out.println( balance + "\t" + amount );

}

// modified toString() method

public String toString()

{

return "Account: " + accountNumber + "\tName: " + accountHolder +

"\tBalance: " + amount + "\tUse Count: " + useCount;

}

}

### QUESTION 4:

Is this toString() method correct?

**Answer:**

No. The formal parameter amount belongs to the processDeposit method. It cannot be used by any other method.

**Scope of a Formal Parameter**

The **scope** of a formal parameter is the section of code that can "see" the parameter. The scope of a formal parameter is the body of its method. For example, the scope of amount is the body of its method:

class CheckingAccount

{

. . . .

private int balance;

. . . .

public void processDeposit( int amount )

{ // scope of amount starts here

balance = balance + amount ;

// scope of amount ends here

}

// modified toString() method

public String toString()

{

System.out.println( balance + "\t" + amount );

}

}

The toString() method cannot "see" amount because it is outside the scope of amount. The compiler will not compile this modified program.

**QUESTION 5:**

Can the toString() method see the object's instance variables, such as balance?

### Answer:

Yes.

# What Statements See

Statements of a method can see the object's instance variables and the object's other methods. They cannot see the parameters and local variables of other methods. Here is another look at the CheckingAccount class:

class CheckingAccount

{

. . . .

private int balance;

. . . .

public void processDeposit( int amount )

{ // scope of amount starts here

balance = balance + amount ;

// scope of amount ends here

}

public void processCheck( int amount )

{ // scope of amount starts here

int charge;

incrementUse();

if ( balance < 100000 )

charge = 15;

else

charge = 0;

balance = balance - amount - charge ;

// scope of amount ends here

}

}

Two methods are using the same identifier, amount, for two different formal parameters. Each method has its own formal parameter completely separate from the other method. This is OK. The scopes of the two parameters do not overlap so the statements in one method cannot "see" the formal parameter of the other method.

For example the statement

balance = balance - amount - charge ;

from the second method can only see the formal parameter of that method. The scope of the formal parameter of the other method does not include this statement.

Of course, the formal parameters of any one method must all have different names.

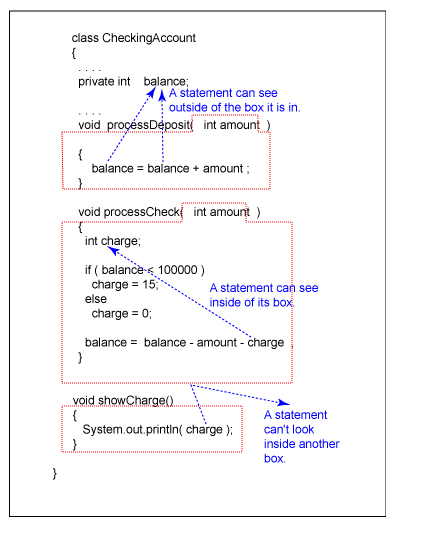
### QUESTION 6:

Could the two formal parameters (of the two methods) named amount be of different types? (For example, could one be an int and the other a long?)

**nswer:**

Yes — since each can only be seen in the body of its method, each can be declared to be of any type.

**One-way Glass**



It is sometimes useful to visualize methods as being surrounded by a box of "one-way glass." A method can see local variables and parameters that are inside the box. A method can look out through the glass that surrounds it. But no outsider can see into the box.

The picture shows the one-way glass for the example program. The red lines show the box of one-way glass surrounding each method. The method can see out of its box (for example each method can see the instance variable balance) but other methods can't see from the outside into the box of one-way glass.

In processDeposit() the statement can "see" the variable balance declared as a instance variable. It can also see the parameter amount that is inside its box.

The method showCharge() is defective because it contains a statement that attempts to look inside the box that surrounds processCheck().

The names of formal parameters (such as amount) and local variables (such as charge) are only visible from inside the glass box. However, the number and type of actual parameters required for each method is known by outsiders.

**QUESTION 7:**

Is the name of a method inside or outside the glass box?

**Answer:**

Outside — so it can be "seen" and used by other methods.

**Assigning to a Parameter**

A parameter is a "one-way message" that the caller uses to send values to a method.

Within the body of a method, a parameter is used just like any variable. It can be used in arithmetic expressions, in assignment statements, and so on.

However, changes made to the parameter do not have any effect outside the method body. A parameter is a local copy of whatever value the caller passed into the method. Any changes made to it affect only this local copy. For example:

class CheckingAccount

{

. . . .

private int balance;

public void processCheck( int amount )

{

int charge;

if ( balance < 100000 )

charge = 15;

else

charge = 0;

balance = balance - amount - charge ;

// change the local copy of the value in "amount"

amount = 0 ;

}

}

class CheckingTester

{

public static void main ( String[] args )

{

CheckingAccount act;

int check = 5000;

act = new CheckingAccount( "123-345-99",

"Wanda Fish", 100000 );

System.out.println( "check:" + check ); // prints "5000"

// call processCheck with a copy of the value 5000

act.processCheck( check );

System.out.println( "check:" + check ); // prints "5000" --- "check" was not changed

}

}

The formal parameter amount is the name used for the value 5000 that processCheck() has been sent. That method changes the value held in amount, but this has no effect on any other variable.

**QUESTION 8:**

Say that main() called the method with an integer literal:

act.processCheck( 7000 ); // call processCheck with the value 7000

Is this OK?

### Answer:

Is this OK? Sure. The caller has sent the value 7000 to the method. Inside the method, that value is held in the parameter amount. At the end of the method, the statement

amount = 0;

puts a zero into the parameter, but does not affect anything in main()

# Local Variables

A **local variable** is a variable that is declared inside of the body of a method. It can be seen only by the statements that follow its declaration inside of that method.

The **scope** of a local variable starts where it is declared and ends at the end of the block that it is in. Recall that a block is a group of statements inside of braces, {}.

For example, charge  of  processCheck is a local variable:

class CheckingAccount

{

. . . .

private int balance;

public void processCheck( int amount )

{

int charge; // scope of charge starts here

incrementUse();

if ( balance < 100000 )

charge = 15;

else

charge = 0;

balance = balance - amount - charge ; // scope of charge ends here

}

}

The local variable charge is used to hold a temporary value while something is being calculated. Local variables are not used to hold the permanent values of an object's state. They have a value only during the brief amount of time that a method is active.

### QUESTION 9:

Is the scope of a local variable always the entire body of a method?

### Answer:

No — the scope starts where the variable was declared and continues to the end of its block. Sometimes a local variable is declared in the middle of a block, close to the statement which first uses it.

# Can't use the Same Name in the Same Scope

It is a mistake to use the same identifier twice in the same scope. For example, the following is a mistake:

class CheckingAccount

{

. . . .

private int balance;

public void processCheck( int amount )

{

int amount;

incrementUse();

if ( balance < 100000 )

amount = 15; // which amount ???

else

amount = 0; // which amount ???

balance = balance - amount ; // which amount ???

}

}

The scope of the formal parameter amount overlaps the scope of the local variable (also named amount), so this is a mistake. This is a different situation than a previous example where two methods used the same name for their formal parameters. In that situation the scopes did not overlap and there was no confusion.

### QUESTION 10:

Can the same identifier be used as a name for a local variable in two different methods?

**Answer:**

Yes — the scopes will not overlap so there will be two local variables, one per method, each with the same name.

**Instance Variable and Local Variable   
with Same Name**

Although it is usually a bad idea, you can declare a formal parameter or a local variable with the same name as one of the instance variables. For example,

class CheckingAccount

{

. . . .

private int balance;

. . . .

public void processDeposit( int amount )

{

int balance = 0; // New declaration of balance.

balance = balance + amount ; // This uses the local variable, balance.

}

}

This is not a syntax error (although it will probably result in a logic error, a bug). The compiler will compile this code without complaint. The second declaration of balance (the one in red) creates a local variable for the processDeposit method. The scope of this variable starts with its declaration and ends at the end of the block (as with all local variables). So the next statement uses the local variable, not the instance variable.

When this modified method is called, it will add amount to the local variable balance, and then return to the caller. The local variable will no longer hold a value after the method has returned. The instance variable will not have been changed.

**Hint:**   Think of statements as looking "upward" from their own location to find each of their variables. They can look outside of their "glass box" in any direction if they fail to find a variable inside their own method.

It is almost always a mistake to use the same name for an instance variable and for a local variable. But it is not a syntax error, so the compiler will not warn you of impending doom.

**QUESTION 11:**

Examine this modification:

class CheckingAccount

{

. . . .

private int balance;

. . . .

public void processDeposit( int amount )

{

int balance = 0; // New declaration of balance.

this.balance = balance + amount ; // ??????

}

}

Does the method change the instance variable balance ?

### Answer:

Yes. The variable that follows the reserved word this is an instance variable (part of "this" object).

# Method Overloading

**Overloading** is when two or more methods of a class have the same name but have different parameter lists. When a method is called, the correct method is picked by matching the actual parameters in the call to the formal parameter lists of the methods.

Review: another use of the term "overloading" is when an operator calls for different operations depending on its operands. For example + can mean integer addition, floating point addition, or string concatenation depending on its operands.

Say that two processDeposit() methods were needed:

* One for ordinary deposits, for which there is no service charge.
* One for other deposits, for which there is a service charge.

class CheckingAccount

{

. . . .

private int balance;

. . . .

public void processDeposit( int amount )

{

balance = balance + amount ;

}

public void processDeposit( int amount, int serviceCharge )

{

balance = balance + amount - serviceCharge;

}

}

The above code implements these requirements. Here is an example main() method that uses both methods:

class CheckingAccountTester

{

public static void main( String[] args )

{

CheckingAccount bobsAccount = new CheckingAccount( "999", "Bob", 100 );

bobsAccount.processDeposit( 200 ); // statement A

bobsAccount.processDeposit( 200, 25 ); // statement B

}

}

There are two method calls, and two methods to pick from. A method call invokes to the method that matches it with both its name and its parameter list.

### QUESTION 12:

Examine main().

Which method, processDeposit(int) or processDeposit(int, int) does each statement call?

1. statement A calls
2. statement B calls

### Answer:

1. statement A calls processDeposit(int)
2. statement B calls processDeposit(int, int)

# Method Signature

When several methods have the same name, only one is picked by a method call:

The types of the actual parameters in a method call are matched with the types of the formal parameters of the methods. If an exact match cannot be made, then the actual parameters are converted to types that match the formal parameters if this can be done without potential loss of information.

For example, the call

bobsAccount.processDeposit( 200, 25 ); //statement A

matches this method declaration:

public void processDeposit( int amount, int serviceCharge )

because the number and types of the actual parameters matches the number and types of the formal parameters.

The **signature** of a method is:

* Its name.
* The number and types of its parameters, in order.

The signatures of the methods in a class must be unique. For example, the signatures of the two processDeposit methods are:

* processDeposit( int )
* processDeposit( int, int )

The names of the parameters are not part of the signature because parameter names are not visible outside of their scope.

The return type is not part of the signature. The visibility modifier is not part of the signature.

### QUESTION 13:

Say that a class has the following two methods:

float chargePenalty( int amount ) { ... }

int chargePenalty( int penalty ) { ... }

Do these methods have unique signatures?

### Answer:

No.

# Return Type is Not Part of the Signature

The names of the formal parameters are not part of the signature, nor is the return type. The signatures of the two methods are:

chargePenalty( int )

chargePenalty( int )

It might seem strange that the return type is not part of the signature. But a potentially confusing situation is avoided by keeping the return type out of the signature. Say that there were two methods that differ only in their return type:

float chargePenalty( int amount ) { ... }

int chargePenalty( int penalty ) { ... }

and that main used one of the methods:

class CheckingAccountTester

{

public static void main( String[] args )

{

CheckingAccount bobsAccount = new CheckingAccount( "999", "Bob", 100 );

double result = bobsAccount.chargePenalty( 60 );

}

}

Which method should be called?

Either method matches the statement, since both have and int parameter and both return a type that can be converted to double. (Both int and float can be converted to double.)

To avoid confusion in situations like this, the return type is not counted as part of the signature.

### QUESTION 14:

Say that a class has these two methods:

public void changeInterestRate( double newRate ) { ... }

public void changeInterestRate( int newRate ) { ... }

Do these methods have unique siqnatures?

### Answer:

Yes. The names of the formal parameters do not have to be unique.

**Fill in the Blanks**

This exercise reviews the "scope" of variables and parameters. The *scope* of a variable or formal parameter is the section of code that can "see" (can use) the parameter.

**The scope of an instance variable includes each method body (list of statements) and each constructor body.**

1.   In the following program skeleton, click on each button where it would be correct to have the statement: target = 25

In other words, click on each button where the variable target is in scope.

Top of Form

class ScopeEg1

{

int target;

ScopeEg1()

{

;

. . . .

}

void aMethod()

{

;

. . . .

}

void bMethod()

{

;

. . . .

}

}

class AnotherClass

{

int sum;

AnotherClass()

{

;

. . . .

}

void anotherMethod()

{

;

. . . .

}

void someMethod()

{

;

. . . .

}

}

class TesterClass

{

public static void main (String[] args )

{

;

}

}

Bottom of Form

**"Outsiders" can access instance variables of an object using "dot notation" unless the instance variable is *private* (or has default access and is in a different package...but ignore this for now.)**

2.   In the following program skeleton, click on each button where it would be OK to have the statement: target = 25;

Top of Form

class ScopeEg2

{

int target;

. . . .

}

class AnotherClass

{

int sum;

ScopeEg2 first = new ScopeEg2() ;

void anotherMethod()

{

first . ;

. . . .

}

void someMethod()

{

first . ;

. . . .

}

}

class TesterClass

{

public static void main (String[] args )

{

first . ;

}

}

Bottom of Form

3.   In the following program skeleton, click on each button where it would be OK to have the statement target = 25

Top of Form

class ScopeEg3

{

**private** int target;

ScopeEg3()

{

;

. . . .

}

void aMethod()

{

;

. . . .

}

void bMethod()

{

;

. . . .

}

. . . .

}

class AnotherClass

{

int sum;

ScopeEg3 first = new ScopeEg3() ;

void anotherMethod()

{

first . ;

. . . .

}

void someMethod()

{

first . ;

. . . .

}

}

class TesterClass

{

public static void main (String[] args )

{

ScopeEg3 second = new ScopeEg3() ;

second . ;

}

}

Bottom of Form

**Formal parameters can only be seen by the body of their own method.**

4.   Click on each button where it would be OK to have the statement System.out.println( data );

Top of Form

class SomeClass

{

int sum;

void aMethod( int data )

{

;

. . . .

}

void bMethod()

{

;

. . . .

}

}

class TesterClass

{

SomeClass some;

public static void main (String[] args )

{

some = new SomeClass();

some.aMethod( 99 );

;

}

}

Bottom of Form

**It is OK for formal parameters in two different methods to use the same identifier.**

5.   In the following program skeleton, click on each button where it would be OK to have the statement sum = data ;

Top of Form

class SomeClass

{

int sum;

void aMethod( int data )

{

;

. . . .

}

void bMethod( int data )

{

;

. . . .

}

void cMethod( int value )

{

;

. . . .

}

}

Bottom of Form

**A local variable can only be seen in the body of its method by statements following its declaration. It is OK for local variables in different methods to use the same name.**

6.   Click on each button where it would be OK to have the statement value = 5;

Top of Form

class SomeOtherClass

{

int sum;

void aMethod( int data )

{

int value;

;

. . . .

}

void bMethod( int data )

{

;

. . . .

}

void cMethod[[egg]]

{

. . . .

;

int value;

. . . .

}

void dMethod[[egg]]

{

double value;

;

. . . .

}

}

Bottom of Form

**If a local variable has the same name as an instance variable the local variable will be the one seen by the statements in its method that follow its declaration.**(Although it is correct syntax to have both local and instance variables use the same name, it is probably a bad idea since it confuses humans.)

7.   Decide if each statements sets the **instance variable** sum or the **local variable** sum.

Top of Form

class YetOtherClass

{

int sum; // the instance variable

void aMethod( int data )

{

sum = data ; ;

. . . .

}

void bMethod( int data )

{

int sum; // a local variable

sum = data ; ;

. . . .

}

void cMethod[[egg]]

{

. . . .

sum = 23 ; ;

int sum;

. . . .

}

}

Bottom of Form

**If a local variable has the same name as an instance variable and you want to specify the instance variable, use *this.***

8.   In the following program skeleton, decide if each statements sets the **instance variable** sum or the **local variable** sum.

Top of Form

class AfurtherClass

{

int sum; // the instance variable

void aMethod( int data )

{

int sum; // a local variable

this.sum = data ; ;

. . . .

}

void bMethod( int data )

{

int sum; // a local variable

sum = data ; ;

. . . .

}

}

Bottom of Form

**If a *parameter* has the same name as an instance variable and you want to specify the instance variable, use *this.***This is often done with constructors, where it is probably *less* confusing to use the same name for both.

9.   Decide if each statements sets the **instance variable** sum or the **parameter** sum.

Top of Form

class AfurtherClass

{

int sum; // instance variable

AfurtherClass( int sum ) // constructor

{

this.sum = sum ; ;

. . . .

}

void bMethod( int sum )

{

sum = 32 ; ;

. . . .

}

}

Bottom of Form

**An "outsider" can change a *private* instance variable of an object by using an access method of the object (if there is one.)**

10.   In the following program skeleton, click on those buttons next to statements that change *sum*.

Top of Form

class SimpleClass

{

private int sum;

SimpleClass( int s )

{

sum = s ; ;

. . . .

}

void setSum ( int s )

{

sum = s ; ;

. . . .

}

}

class TesterClass

{

public static void main ( String[] args )

{

SimpleClass sim = SimpleClass( 34 ); ;

sim.sum = 77 ; ;

sim.setSum( 14 ) ; ;

}

}

# Object Parameters

Parameters are used to pass values from a caller into a method. This chapter examines parameters that are object references.

#### Chapter Topics:

* Review of parameters
* Call by value with primitive data types
* Object references as parameters
* Immutable objects as parameters
* Changeable objects as parameters

### QUESTION 1:

(Reviewsmile What is a formal parameter?

What is an actual parameter?

### Answer:

* **formal parameter** — the identifier used in a method to stand for the value that will be passed into the method by a caller.
* **actual parameter** — the actual value that is passed into the method by a caller.

# Call by Value

The type of parameter passing that Java uses is called **call by value**. Some programming languages use other methods of parameter passing, although call by value is the most common.

This is how call by value works:

1. When the caller invokes a method, the caller provides a list of values (the actual parameters) in the parameter list.
2. When the invoked method starts running, these values are copied to the formal parameters.
3. The invoked method uses the formal parameters to access these copied values.
4. Any change that the method makes to the value held in a formal parameter changes only that copy.
5. The invoked method cannot use a formal parameter to send a value back to the caller.

### QUESTION 2:

(Reviewsmile What is a primitive data type?

**nswer:**

A *primitive data type* is one of the eight fundamental methods used to represent data that is built into Java.

**Primitive Data Types as Parameters**

The eight primitive data types are:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| byte | short | int | long | float | double | char | boolean |

The word **primitive** means "a fundamental piece that is used to create other, larger parts." So far we have been using parameters with primitive data types. Here is a tiny program that reviews this:

class SimpleClass

{

public void print( int x )

{

System.out.println("Value of the parameter: " + x );

}

}

class SimpleTester

{

public static void main ( String[] args )

{

int var = 7;

SimpleClass simple = new SimpleClass();

System.out.println("First value of the local var: " + var );

simple.print( var );

System.out.println("Second value of the local var: " + var );

}

}

**QUESTION 3:**

What is the output of the program?

Top of Form

First value of the local var:

Value of the parameter:

Second value of the local var:

### Answer:

First value of the local var: 7

Value of the parameter: 7

Second value of the local var: 7

# Changes to the Formal Parameter do not affect the Caller

Here is the program again, with a slight change. Now the invoked method print() makes a change to its copy of the value in its formal parameter x.

class SimpleClass

{

public void print( int x )

{

System.out.println("First value of the parameter: " + x );

x = 100; // local change to the formal parameter

System.out.println("Second value of the parameter: " + x );

}

}

class SimpleTester

{

public static void main ( String[] args )

{

int var = 7;

SimpleClass simple = new SimpleClass();

System.out.println("First value of the local var: " + var );

simple.print( var );

System.out.println("Second value of the local var: " + var );

}

}

Recall that this is using call by value which means parameters are used to pass values into a method, but not used to pass anything back to the caller.

### QUESTION 4:

Now what is the output of the program?

Top of Form

First value of the local var:

First value of the parameter:

Second value of the parameter:

Second value of the local var:

### Answer:

First value of the local var: 7

First value of the parameter: 7

Second value of the parameter: 100

Second value of the local var: 7

# Returning a Value

Once a value has been copied into an invoked method the invoked method can use the copy or change the copy. But these changes do not affect any variable inside the caller. How can a method send a value back to the caller? Examine the following:

class SimpleClassTwo

{

public int twice( int x )

{

return 2\*x;

}

}

class SimpleTesterTwo

{

public static void main ( String[] args )

{

int var = 7;

int result = 0;

SimpleClassTwo simple = new SimpleClassTwo();

System.out.println("First value of result: " + result );

result = simple.twice( var );

System.out.println("Second value of result: " + result );

}

}

To return a single value to the caller, an invoked method can use the return statement along with the value to be returned.

### QUESTION 5:

Now what is the output of the program?

Top of Form

First value of the result:

Second value of the result:

### Answer:

First value of the result: 0

Second value of the result: 14

# Object References as Parameters

Object references can be parameters. Call by value is used, but now the value is an object reference. This reference can be used to access the object and possibly change it. Here is an example program:

class ObjectPrinter

{

public void print( String st )

{

System.out.println("Value of parameter: " + st );

}

}

class OPTester

{

public static void main ( String[] args )

{

String message = "Only One Object" ;

ObjectPrinter op = new ObjectPrinter();

System.out.println("First value of message: " + message );

op.print( message );

System.out.println("Second value of message: " + message );

}

}

In this program, the local variable message contains a reference to a String object. A **copy** of that reference is made in the formal parameter st of the print() method. The object is not copied.

### QUESTION 6:

What is the output of the program?

First value of message:

Value of parameter:

Second value of message:

**Answer:**

First value of message: Only One Object

Value of parameter: Only One Object

Second value of message: Only One Object

**Only One Object**

The program works as you expect. The diagram shows what is happening. The main() method creates a String object that contains the characters "Only One Object." A reference to this object is held in the reference variable message.

A reference to an object is a way to find the object in main memory. If a method has a reference to an object, then in can use that object.

|  |
| --- |
|  |

Now an ObjectPrinter object is created and a reference to it is placed in op. In the statement op.print(message), the reference to the object is passed as the value of the parameter. This is just like call by value with a primitive data type, but now the value is a reference.

The invoked method print() uses its formal parameter st to find the object, and print out the object's data.

**QUESTION 7:**

If print() changes the value held in st, will this change the actual object?

### Answer:

No. Changing the reference will not change the object.

# Changing a Reference Parameter

Here is an altered program in which the print() method changes the value held in its formal parameter.

class ObjectPrinter2

{

public void print( String st )

{

System.out.println("First value of parameter: " + st );

st = "Hah! A second Object!" ;

System.out.println("Second value of parameter: " + st );

}

}

class OPTester2

{

public static void main ( String[] args )

{

String message = "Original Object" ;

ObjectPrinter op = new ObjectPrinter();

System.out.println("First value of message: " + message );

op.print( message );

System.out.println("Second value of message: " + message );

}

}

### QUESTION 8:

What is the output of the program?

Top of Form

First value of message:

First value of parameter:

Second value of parameter:

Second value of message:

**Answer:**

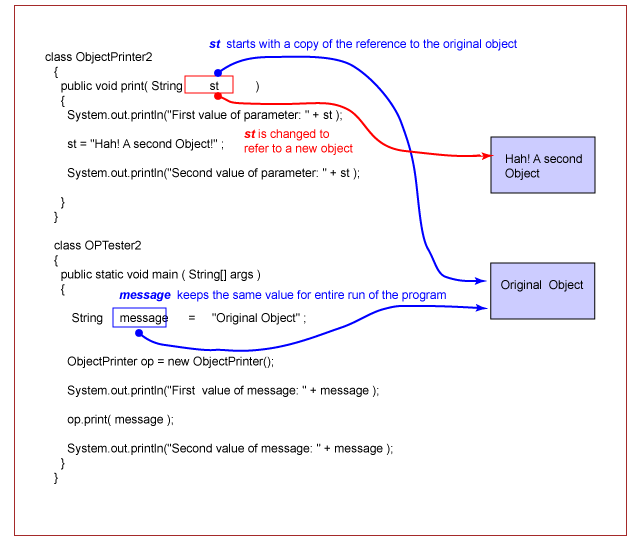
First value of message: Original Object

First value of parameter: Original Object

Second value of parameter: Hah! A second Object!

Second value of message: Original Object

**Revised Picture**



In the revised program, the print() method created a new object and placed a reference to it into the parameter st. As long as you know what you are doing, this is OK.

Each color of line in the picture represents a different reference value. The picture shows the print() method changing the value in the formal parameter so that it refers to a second object.

|  |
| --- |
|  |

Neither the original object, nor the variable message of the main() method are changed. The print() method can use its formal parameter just like any variable, including changing the value it holds.

**QUESTION 9:**

Could print() use the reference to the original String object to change the contents of that object?

**Answer:**

No — because String objects are *immutable*. Not even the main() method can change the original object.

**Immutable Strings**

It is good that String objects are immutable (they can't be changed) because the main() method can be sure that the *message* is completely under its control. Although the print() method gets a copy of the reference, it can't change the original object.

Not all objects are immutable. For example, in the following, MyPoint objects have public instance variables x, and y. These can be changed by any method that has a referece to the object.

class MyPoint

{

public int x=3, y=5;

public void print()

{

System.out.println("x = " + x + "; y = " + y );

}

}

class PointTester

{

public static void main ( String[] args )

{

MyPoint pt = new MyPoint();

pt.print();

pt.x = 45; pt.y = 83;

pt.print();

}

}

**Important:** Public instance variables of objects can be changed by any method that has a reference to the object.

(If an instance variable is neither public nor private it can be changed by a method that is in the same *package*. For now, all of our code is in the same package, so the effect is the same as if it were public.)

The main() method uses the default constructor of class MyPoint. This is the constructor you get automatically if you do not define one yourself.

**QUESTION 10:**

Top of Form

What is the output of the program?

x = y =

x = y =

**Answer:**

x = 3; y = 5

x = 45; y = 83

**Mutable MyPoint**

As the example program shows, a MyPoint object can be changed by any method that has a reference to it.

**Note:** If a reference to a MyPoint object is passed as a parameter, then the invoked method can use that reference to change the public instance variables of the object.

Here is the example program with another class:

|  |
| --- |
|  |

The picture shows the situation when the twice() method of the PointDoubler object is active. The method has a reference to the single MyPoint object that is present and can alter the data in that object.

**QUESTION 11:**

Top of Form

What is the output of the program?

x = y =

Enter PointDoubler

x = y =

x = y =

Leave PointDoubler

x = y =

### Answer:

x = 3; y = 5

Enter PointDoubler

x = 3; y = 5

x = 6; y = 10

Leave PointDoubler

x = 6; y = 10

# Still Call by Value

Here are some facts:

* Paramater passing is always call by **value**

.

* If a method's parameter is a **primitive** data type, the method can change its own value for the parameter, but the change has no effect elsewhere.
* However, if a method's parameter is a **reference to an object**, the instance variables of the object can be changed (if they are public).

These facts are consistent with call by value. The "value" is a reference to the object. The invoked method has its local copy of this reference and can't change the value of the reference held by the caller. However, the invoked method can change the object.

Of course, even if a method has a reference to an object, the object can be changed only if the object allows changes to be made (either with public instance variables or through access methods).

### QUESTION 12:

Look at the definition of the MyPoint class. Think of a way to make MyPoint objects immutable.

### Answer:

The two instance variables can be made private.

# MyPoint with better Encapsulation

Here is a revised version of the program. Now MyPoint objects are immutable. Methods can't change the object, even if they have a reference to it. All they can do is call the object's public methods.

class ImmutablePoint

{

private int x, y;

public ImmutablePoint( int px, int py )

{

x = px; y = py;

}

public void print()

{

System.out.println("x = " + x + "; y = " + y );

}

}

class PointPrinter

{

public void print( ImmutablePoint p )

{

p.print(); // call a public method

p.x = 77 ; // WRONG! can't do this

}

}

class PointTester

{

public static void main ( String[] args )

{

ImmutablePoint pt = new ImmutablePoint( 4, 8 );

pt.print(); // call a public method

pt.x = 88; // WRONG! can't do this

PointPrinter pptr = new PointPrinter();

pptr.print( pt ); // ca

}

}

Since ImmutablePoint objects are immutable, a constructor is needed to initialize instance variables to their permanent values.

### QUESTION 13:

(Thought Questionsmile Would it be possible to write a PointDoubler class for ImmutablePoint objects?

### Answer:

No — any such class could not gain access to the instance variables it needs to double.

**Instructions:** For each question, choose the single best answer. Make your choice by clicking on its button. You can change your answers at any time. When the quiz is graded, the correct answers will appear in the box after each question.

Top of Form

1. What is an *actual parameter*?

 **A.** The identifier used in a method to stand for the value that is passed into a method by a caller.

 **B.** The value that is passed into a method by a caller.

 **C.** A variable used to control a counting loop.

 **D.** The value that is returned by a method.



Bottom of Form

Top of Form

2. What type of parameter passing is used by Java?

 **A.** Call by value.

 **B.** Call by object.

 **C.** Call by parameter.

 **D.** Call waiting.



Bottom of Form

Top of Form

3. If a method assigns a new value to a **primitive** parameter, will this have any effect on its caller?

 **A.** No, because all the method has is a copy of its caller's values.

 **B.** No, because the method is not allowed to change the value of any parameter.

 **C.** Yes, the change will affect the matching value in the caller.

 **D.** Yes, because the method and its caller share the same section of memory.



Bottom of Form

Top of Form

4. If a method assigns a new *object* to an **object reference** parameter, will this have any effect on its caller?

 **A.** No, because this will not affect any object that the caller can reference.

 **B.** No, because this is not a legal operation.

 **C.** Yes, because now the caller can reference the new object.

 **D.** Yes, because the new object will replace one of the caller's objects.



Bottom of Form

Top of Form

5. If a method assigns a new value to a *member* of an object it can access through an **object reference** parameter, will this have any effect on its caller?

 **A.** No, because it only has a copy of the object.

 **B.** No, because it is not allowed to do this.

 **C.** Yes, this will change part of the object that both it and the caller are referring to.

 **D.** Yes, the caller will now get a new object.



Bottom of Form

Top of Form

6. How can a method communicate a primitive value back to the caller?

 **A.** It can assign the value to one of its parameters.

 **B.** It can return the value to the caller using a *return statement*.

 **C.** A method can never communicate with its caller by any means.

 **D.** It can call its caller with the correct value.



Bottom of Form

Top of Form

7. If a method has a reference to an *immutable object*, such as a *String* object, can the method make a change to the object?

 **A.** No, immutable objects can't be changed by anyone after they have been created.

 **B.** No, only the creator of an immutable object can change it.

 **C.** Yes, if it has a reference to an immutable object it can change it.

 **D.** Yes, but it must use the special "+" operator to do so.



Bottom of Form

Top of Form

8. Can a programmer (you) design a class whose objects are *immutable*?

 **A.** No, only a few predefined Java classes like *String* are immutable.

 **B.** No, all pre-defined classes are immutable, and all programmer defined classes are not.

 **C.** Yes, by making all instance variables of the class *private*.

 **D.** Yes, by making all instance variables and member methods *private*.



Bottom of Form

Top of Form

9. How many references can there be to a single object?

 **A.** Only one.

 **B.** Two: one in a caller and one in the called method.

 **C.** Three: the original reference, and one reference each for a formal and an actual parameter.

 **D.** There can be any number of references, held in any number of variables and parameters (as long as they are of the correct type.)



Bottom of Form

Top of Form

10. Examine the following:

String mess = "Hello" ;

mess = mess + " World" ;

What does the second statement do?

 **A.** It adds the characters *World* to the *String* object referenced by *mess*.

 **B.** It creates a new *String* object based on the original object referenced by *mess* and another *String* object containing *World*.

 **C.** It alters the immutable object referenced by *mess*.

 **D.** It is illegal because it attempts to alter an immutable object.



# Objects that Contain Objects

# 

One of the ideas of object oriented programming is to have software objects imitate "real world" objects. Most real world objects are made up of smaller objects. For instance, a bicycle is an object, and it is an assembly of several objects — frame, wheels, gears, handle bars, and other parts. Some of those parts are, in turn, made up of yet smaller parts. A wheel is made of a rim, a tire, an inner tube, a hub, and many spokes.

As with real world objects, software objects are often composed of smaller software objects. The syntax of Java makes this easy to do. This chapter shows how to do it.

#### Chapter Topics:

* Defining a class that contains smaller classes
* Example: Making a Fleet object out of Car objects
* Example: Defining Fleet methods in terms of Car methods

### QUESTION 1:

Could we continue to break down a bicycle into smaller and smaller objects?

Bottom of Form

**Answer:**

Yes. The rim, for instance, is made of yet smaller parts.

**Class Car**

Part of the skill of object oriented programming is in designing the right *software objects* to match the *real world objects* of the problem, and in deciding when to quit. It would (usually) make little sense to continue analyzing the bicycle until you were looking at atoms. On the other hand, sometimes very fine detail is desirable. A good flight simulator program (for example) is realistic because its airplane objects are built up from many small objects. A crude simulation uses fewer objects.

Look at the definition for Car from a few chapters ago:

class Car

{

// data

int startMiles; // Starting odometer reading

int endMiles; // Ending odometer reading

double gallons; // Gallons of gas used between the readings

// constructor

Car( int first, int last, double gals )

{

startMiles = first ;

endMiles = last ;

gallons = gals ;

}

// methods

double calculateMPG()

{

return (endMiles - startMiles)/gallons ;

}

}

**QUESTION 2:**

Top of Form

Is the Car class made up of smaller software objects?   Click here for a

What are the *instance variables* of class Car?   Click here for a

* 
* 
* 

What *method* is part of class Car?   Click here for a

* 

**Answer:**

Is the Car class made up of smaller software objects?

* No. It consists of some primitive (non-object) data and one method.

What are the *instance variables* of class Car?

* startMiles
* endMiles
* gallons

What is the *method* of class Car?

* calculateMPG()

**Holding the State of an Object**

The state of an object consists of the values held by its instance variables. These may change during the lifetime of the object. Here is a main() program that constructs a Car object.

class MpgTester

{

public static void main ( String[] args )

{

Car myCar = new Car( 12000, 12340, 12.3 );

. . . . . .

}

}

When the object is constructed, its state is initialized to:

* startMiles = 12000
* endMiles = 12340
* gallons = 12.3

The object referenced by myCar holds these values as long as it exists.

**QUESTION 3:**

Could a collection of several cars be regarded as an object?

**Answer:**

Yes. A fleet of cars consists of several cars.

**Car Fleet**

Consider a fleet consisting of two cars: a town car and a sports utility vehicle. Think of the fleet as a single object composed of two objects. Remember that an object has (i) identity, (ii) state, and (iii) behavior. Is this true for a fleet of cars?

1. **Identity:** Yes — my fleet is different from your fleet.
2. **State:** Yes — the odometer readings of each car is part of the state of the fleet.
3. **Behavior:** Yes — the fleet can have a method to compute average MPG for the fleet (and can have other methods if we care to write them.)

Here is a skeleton of the Fleet class, along with the definition of Car and a testing class:

class Fleet

{

// data

Car town; // the town car of the fleet

Car suv; // the sports utility vehicle of the fleet

// constructor

// method

}

class Car // unchanged from above

{

. . .

}

class FleetTester

{

Fleet myCars;

. . .

}

The definition of class Car is the same as above. It does not change, even though Car objects are now used as part of a larger object. The (not yet finished) definition of Fleet says that a fleet consists of two cars.

**QUESTION 4:**

Does the definition of class Fleet include the definition of class Car?

**Answer:**

No. However, the definition for Fleet says that a Fleet object has two variables that refer to Car objects.

**Instance Variables for Fleet**

The state of a Fleet object is held in its two instance variables. These variables refer to Car objects. The documentation for Fleet might look like this:

**Fleet**

A class that holds two Car objects.

**Constructors**

Fleet( int Car1StartOdo, int Car1EndingOdo, double Car1Gallons,

int Car2StartOdo, int Car2EndingOdo, double Car2Gallons )

Creates a new instance of a Fleet object with the starting and ending odometer readings and the number of gallons of gas consumed for each car.

**Methods**

double calculateMPG()

Calculates and returns the average miles per gallon for the fleet.

The constructor builds the two cars of the fleet. Each car needs three initial values. So the constructor for Fleet has a total of six initial values. (There are other, more elegant, ways to do this, but let us do it this way for now.) Here is a short   main()   that constructs a fleet:

Top of Form

class FleetTester

{

public static void main ( String[] args)

{

Fleet myCars = new Fleet(  );

}

}

Bottom of Form

**QUESTION 5:**

Fill in the blanks so that the Fleet object looks like this:

1. The first car in the fleet has odometer readings of 1000, 1234, and gallons of 10
2. The second car in the fleet has odometer readings of 777, 999, and gallons of 20

**Chapter 35 Fill in the Blanks**

This exercise will give you practice in composing objects out of other objects. First you will define the *Jam* class. Objects of this class represent jars of fruit preserves. Then several these jars will be placed in a *Pantry*.

1.       **Design of the Jam Class**.     Objects of this class will represent a jar of fruit preserves such as your grandmother made every Fall. Think of three values you would like to know about a jar of jam.

Top of Form

Bottom of Form

Now think of some methods that apply to a *Jam* object. It is almost always a good idea to have a way to print the data of an object:

Top of Form

Bottom of Form

This is a somewhat silly example, and it would not be unusual if you thought of different ways to fill these blanks.

2.      **Document the class**.     Now fill in the blanks for the documentation of the class:

Top of Form

**class**

A class that models a jar of jam.

**Constructors**

Jam ( String , // think of a good parameter name

String , // for each of the three values

int )

**Methods**

// check if the jar is empty

public boolean '">

// remove some jam --- spread an amount of it on toast

public void spread

// print info about the jar

public void print()

Bottom of Form

3.      **Checking the Design**.    To check the design, write a small program that uses the class. Of course, the program can't be compiled and run until the class is written, but you can get a feel for if the class design is sensible by doing this.

Top of Form

class JamTester

{

public static void main ( String[] args )

{

jar = new ( "crab apple", "9/30/99", 12 );

jar.print();

if ( jar. )

System.out.println("Can't spread any jam: jar is empty.");

else

{

jar. ;

System.out.println("Spreading 1 oz. of jam.");

jar.print();

}

}

}

Bottom of Form

You may have some doubts that this is a sensible program. Perhaps it would be a good idea to have the spread() method print out a message each time it is used.

4.      **Fill in Instance Variables**.    Fill in the data type of each instance variable.

Top of Form

class Jam

{

// Instance Variables

contents;

date ;

capacity;

// Constructors

// Methods

}

Bottom of Form

5.      **Complete the Constructor**.    The constructor will initialize the instance variables of the object being constructed.

Top of Form

class Jam

{

// Instance Variables

String contents ; // type of fruit in the jar

String date ; // date of canning

int capacity ; // amount of jam in the jar

// Constructors

Jam( String contents, String date, int size )

{

. contents = contents ;

. date = date ;

= size;

}

// Methods

}

Bottom of Form

6.      **Complete the Methods**.     If the user wants to spread more jam than is available, spread only the amount that is left in the jar.

Top of Form

class Jam

{

// Instance Variables

String contents ; // type of fruit in the jar

String date ; // date of canning

int capacity ; // amount of jam in the jar

// Constructors

Jam( String contents, String date, int size )

{

this . contents = contents ;

this . date = date ;

capacity = size;

}

// Methods

public boolean empty ()

{

return ( == ) ;

}

public void print ()

{

System.out.println ( + " " +

+ " " + + " fl. oz." ) ;

}

public void spread ( int fluidOz)

{

if ( !empty() )

{

if ( fluidOz <= capacity )

{

System.out.println("Spreading " + + " fluid ounces of "

+ contents );

capacity = capacity - ;

}

else

{

System.out.println("Spreading " + + " fluid ounces of "

+ contents );

capacity = ;

}

}

else

System.out.println("No jam in the Jar!");

}

}

Bottom of Form

7.      **Design of the *Pantry* class**.     Let us say that a pantry consists of three jars of jam of any type and size. The methods of a pantry will be to:

1. List the contents of the pantry.
2. To *select* a jar by number
   * Jars will be selected by integers 1, 2, or 3.
3. To *spread* jam from the selected jar.

To use a jar of jam, one must first go to the pantry and select one.

Top of Form

Here is the documentation:

**class**

A class that models pantry full of jam.

**Constructors**

Pantry ( jar1,

jar2,

jar3 )

**Note:** this constructor will take three object references as parameters. The three objects will be the three jars to be kept in the pantry. The objects must already exist before using this constructor.

**Methods**

// print the contents of the pantry

void print()

// select a jar by number

public void select

// spread an amount of jam from the selected jar

public void spread

Bottom of Form

8.      **Instance Variables and constructor**.     Decide on the instance variables. There will be a variable that refers to the currently selected jar of jam. When no jar is selected, it should refer to no object. Outsiders should not be able to directly change the variables.

Top of Form

class Pantry

{

// Instance Variables

private jar1 ;

private jar2 ;

private jar3 ;

private selected ;

// Constructors

Pantry( Jam jar1, Jam jar2, Jam jar3 )

{

. jar1 = jar1 ;

. jar2 = jar2 ;

. jar3 = jar3 ;

selected = ;

}

// Methods

}

Bottom of Form

9.      **Implement *print* method**.     The *print* method will print each of the three jars in the pantry.

Top of Form

class Pantry

{

// Instance Variables

private Jam jar1 ;

private Jam jar2 ;

private Jam jar3 ;

// Constructors

Pantry( Jam jar1, Jam jar2, Jam jar3 )

{

this . jar1 = jar1 ;

this . jar2 = jar2 ;

this . jar3 = jar3 ;

selected = null ;

}

// Methods

public void print()

{

System.out.print("1: "); . print() ;

System.out.print("2: "); . print() ;

System.out.print("3: "); . print() ;

}

}

Be sure that you understand

Bottom of Form

10.      **Implement the *select()* and *spread()* methods**.

Top of Form

class Pantry

{

// Instance Variables

private Jam jar1 ;

private Jam jar2 ;

private Jam jar3 ;

private Jam selected ;

// Constructors

Pantry( Jam jar1, Jam jar2, Jam jar3 )

{

this . jar1 = jar1 ;

this . jar2 = jar2 ;

this . jar3 = jar3 ;

selected = null ;

}

// Methods

public void print()

{

System.out.print("1: "); jar1 . print() ;

System.out.print("2: "); jar2 . print() ;

System.out.print("3: "); jar3 . print() ;

}

// assume that the user entered a correct selection, 1, 2, or 3

public void select( int jarNumber )

{

if ( jarNumber 1 )

selected = ;

else if ( jarNumber 2 )

selected = ;

else

selected = ;

}

// spread the selected jam

public void spread( int oz )

{

. spread( oz ) ;

}

}

Bottom of Form

**Entire Program, with testing class:**     You might wish to copy this program to your editor, save it to a file, and to play with it.

class Jam

{

// Instance Variables

String contents ; // type of fruit in the jar

String date ; // date of canning

int capacity ; // amount of jam in the jar

// Constructors

Jam( String contents, String date, int size )

{

this . contents = contents ;

this . date = date ;

capacity = size;

}

// Methods

public boolean empty ()

{

return ( capacity== 0 ) ;

}

public void print ()

{

System.out.println ( contents + " " + date + " " + capacity + " fl. oz." ) ;

}

public void spread ( int fluidOz)

{

if ( !empty() )

{

if ( fluidOz <= capacity )

{

System.out.println("Spreading " + fluidOz + " fluid ounces of "

+ contents );

capacity = capacity - fluidOz ;

}

else

{

System.out.println("Spreading " + capacity + " fluid ounces of "

+ contents );

capacity = 0 ;

}

}

else

System.out.println("No jam in the Jar!");

}

}

class Pantry

{

// Instance Variables

private Jam jar1 ;

private Jam jar2 ;

private Jam jar3 ;

private Jam selected ;

// Constructors

Pantry( Jam jar1, Jam jar2, Jam jar3 )

{

this . jar1 = jar1 ;

this . jar2 = jar2 ;

this . jar3 = jar3 ;

selected = null ;

}

// Methods

public void print()

{

System.out.print("1: "); jar1 . print() ;

System.out.print("2: "); jar2 . print() ;

System.out.print("3: "); jar3 . print() ;

}

// assume that the user entered a correct selection, 1, 2, or 3

public void select( int jarNumber )

{

if ( jarNumber == 1 )

selected = jar1 ;

else if ( jarNumber == 2 )

selected = jar2 ;

else

selected = jar3 ;

}

// spread the selected jam

public void spread( int oz )

{

selected . spread( oz ) ;

}

}

class PantryTester

{

public static void main ( String[] args )

{

Jam goose = new Jam( "Gooseberry", "7/4/86", 12 );

Jam apple = new Jam( "Crab Apple", "9/30/99", 8 );

Jam rhub = new Jam( "Rhubarb", "10/31/99", 3 );

Pantry hubbard = new Pantry( goose, apple, rhub );

hubbard.print();

hubbard.select(1);

hubbard.spread(2);

hubbard.print();

hubbard.select(3);

hubbard.spread(4);

hubbard.print();

}

}

When you run the program you will be rewarded with the output:

1: Gooseberry 7/4/86 12 fl. oz.

2: Crab Apple 9/30/99 8 fl. oz.

3: Rhubarb 10/31/99 3 fl. oz.

Spreading 2 fluid ounces of Gooseberry

1: Gooseberry 7/4/86 10 fl. oz.

2: Crab Apple 9/30/99 8 fl. oz.

3: Rhubarb 10/31/99 3 fl. oz.

Spreading 3 fluid ounces of Rhubarb

1: Gooseberry 7/4/86 10 fl. oz.

2: Crab Apple 9/30/99 8 fl. oz.

3: Rhubarb 10/31/99 0 fl. oz.

|  |
| --- |
| Chapter 35 Programming Exercises These exercises assume that your have read the review exercise for this chapter. All of the programming exercises are modifications of that program. Start by copying that program into Notepad. Exercise 1 — User Interaction. Modify the PantryTester class so that it carries out a dialog with the user:  Welcome to Mother Hubbard's Pantry!  The jams are:  1: Gooseberry 7/4/86 12 fl. oz.  2: Crab Apple 9/30/99 8 fl. oz.  3: Rhubarb 10/31/99 16 fl. oz.  Enter your selection (1, 2, or 3):  1  Enter amount to spread:  2  Spreading 2 fluid ounces of Gooseberry  The jams are:  1: Gooseberry 7/4/86 10 fl. oz.  2: Crab Apple 9/30/99 8 fl. oz.  3: Rhubarb 10/31/99 16 fl. oz.  Enter your selection (1, 2, or 3):  2  Enter amount to spread:  25  Spreading 8 fluid ounces of Crab Apple  The jams are:  1: Gooseberry 7/4/86 10 fl. oz.  2: Crab Apple 9/30/99 0 fl. oz.  3: Rhubarb 10/31/99 16 fl. oz.  Enter your selection (1, 2, or 3):  2  Enter amount to spread:  9  No jam in the Jar!  The jams are:  1: Gooseberry 7/4/86 10 fl. oz.  2: Crab Apple 9/30/99 0 fl. oz.  3: Rhubarb 10/31/99 16 fl. oz.  Enter your selection (1, 2, or 3):  -1  Good-by  The program should first initialize the jams and the pantry, then....   1. List the available jams. 2. Prompt the user. 3. Input the users selection. 4. Write an error message for an out-of-range selection (and then start over). 5. Prompt the user for an amount. 6. Decrease the amount of the selected jam by the proper amount. 7. Exit when the user selects jam -1, otherwise start over.   Use the Jam and Pantry classes from the Review exercise. To do this:   1. Copy the code at the end of the exercise to an editor, save it, and compile it. 2. You will now have Jam.class, Pantry.class and PantryTester.class in your directory. 3. Create a new file, Exercise1.java that implements the user interaction. 4. Exercise1.java will import Jam.class and Pantry.class. 5. Compile and run Exercise1.java.   Exercise1.java will look like this:    import java.io.\*;  import Jam;  import Pantry;  class Exercise1  {  public static void main ( String[] args )  {  Scan scan = new Scanner( System.in );  String inChars;  Jam goose = new Jam( "Gooseberry", "7/4/86", 12 );  Jam apple = new Jam( "Crab Apple", "9/30/99", 8 );  Jam rhub = new Jam( "Rhubarb", "10/31/99", 16 );  . . . . . . . . . .  }  }  [Click here](http://www.webmastercert.com/moodle/index.html#35)to go back to the main menu. Exercise 2 — New Constructors for the Pantry Class Add new constructors to the Pantry Class: a constructor that takes one Jam parameter, and another that takes two Jam parameters. Unused instance variables in the Pantry object will be set to null.  Now modify the methods of the class to deal with null instance variables: print() will test for null before printing out a Jam. select() will have a return type of boolean. Return true if the selection is available, otherwise false.  Here is a PantryTester2 class that demonstrates the new code:    class PantryTester2  {  public static void main ( String[] args )  {  Jam goose = new Jam( "Gooseberry", "7/4/86", 12 );  Jam apple = new Jam( "Crab Apple", "9/30/99", 8 );  Pantry hubbard = new Pantry( goose, apple );  hubbard.print();  if ( hubbard.select(1) )  hubbard.spread(2);  else  System.out.println("Selection not available");  hubbard.print();  if ( hubbard.select(3) )  hubbard.spread(2);  else  System.out.println("Selection not available");  hubbard.print();  }  }  When it is run, it prints the following:    1: Gooseberry 7/4/86 12 fl. oz.  2: Crab Apple 9/30/99 8 fl. oz.  Spreading 2 fluid ounces of Gooseberry  1: Gooseberry 7/4/86 10 fl. oz.  2: Crab Apple 9/30/99 8 fl. oz.  Selection not available  1: Gooseberry 7/4/86 10 fl. oz.  2: Crab Apple 9/30/99 8 fl. oz.  Of course, complete testing would involve a much larger PantryTester2.  [Click here](http://www.webmastercert.com/moodle/index.html#35)to go back to the main menu. Exercise 3 — replace() method Add more code to the program of exercise 2. Write a method replace( Jam j, int slot ) for the Pantry class that replaces a particular jar of jam in the pantry with the object j. Here is a testing program:    class PantryTester3  {  public static void main ( String[] args )  {  Jam goose = new Jam( "Gooseberry", "7/4/86", 12 );  Jam apple = new Jam( "Crab Apple", "9/30/99", 8 );  Jam rhub = new Jam( "Rhubarb", "10/31/99", 16 );  Pantry hubbard = new Pantry( goose, apple );  hubbard.print();  if ( hubbard.select(3) )  hubbard.spread(2);  else  System.out.println("Selection not available");  hubbard.print();  hubbard.replace( rhub, 3 );  hubbard.print();  if ( hubbard.select(3) )  hubbard.spread(2);  else  System.out.println("Selection not available");  hubbard.print();    }  }  [Click here](http://www.webmastercert.com/moodle/index.html#35)to go back to the main menu. Exercise 4 — MixedFruit Add a method to the Pantry class:  public void mixedFruit()  This method checks that each jar of jam in the pantry has 2 fluid ounces or less, and if so, replaces the first jar of jam with a jar of mixed fruit jam. The amount is the combined amount of the original three jars. The last two jars are set to null. (In other words, this method mixes the jam in all three jars to create a new jar that replaces the old jars.) Modify the testing program to use this new method. Here is sample output of a testing program:    1: Gooseberry 7/4/86 4 fl. oz.  2: Crab Apple 9/30/99 1 fl. oz.  3: Rhubarb 10/31/99 2 fl. oz.  Spreading 2 fluid ounces of Gooseberry  1: Gooseberry 7/4/86 2 fl. oz.  2: Crab Apple 9/30/99 1 fl. oz.  3: Rhubarb 10/31/99 2 fl. oz.  1: Mixed Fruit 7/4/86 5 fl. oz.  [Click here](http://www.webmastercert.com/moodle/index.html#35)to go back to the main menu.  End of the Exercises  -->  -->  [Previous](http://www.webmastercert.com/moodle/mod/book/view.php?id=5495&chapterid=1343)[Exit Book](http://www.webmastercert.com/moodle/course/view.php?id=47#section-2) |

You are logged in as [edris safari](http://www.webmastercert.com/moodle/user/view.php?id=4668&course=47) ([Logout](http://www.webmastercert.com/moodle/login/logout.php?sesskey=6nDHYarlPq))

Bottom of Form

Bottom of Form

Bottom of Form

Bottom of Form

Bottom of Form

Bottom of Form

Bottom of Form

Bottom of Form

Bottom of Form

Bottom of Form

Bottom of Form

Bottom of Form

Bottom of Form

Bottom of Form

Bottom of Form