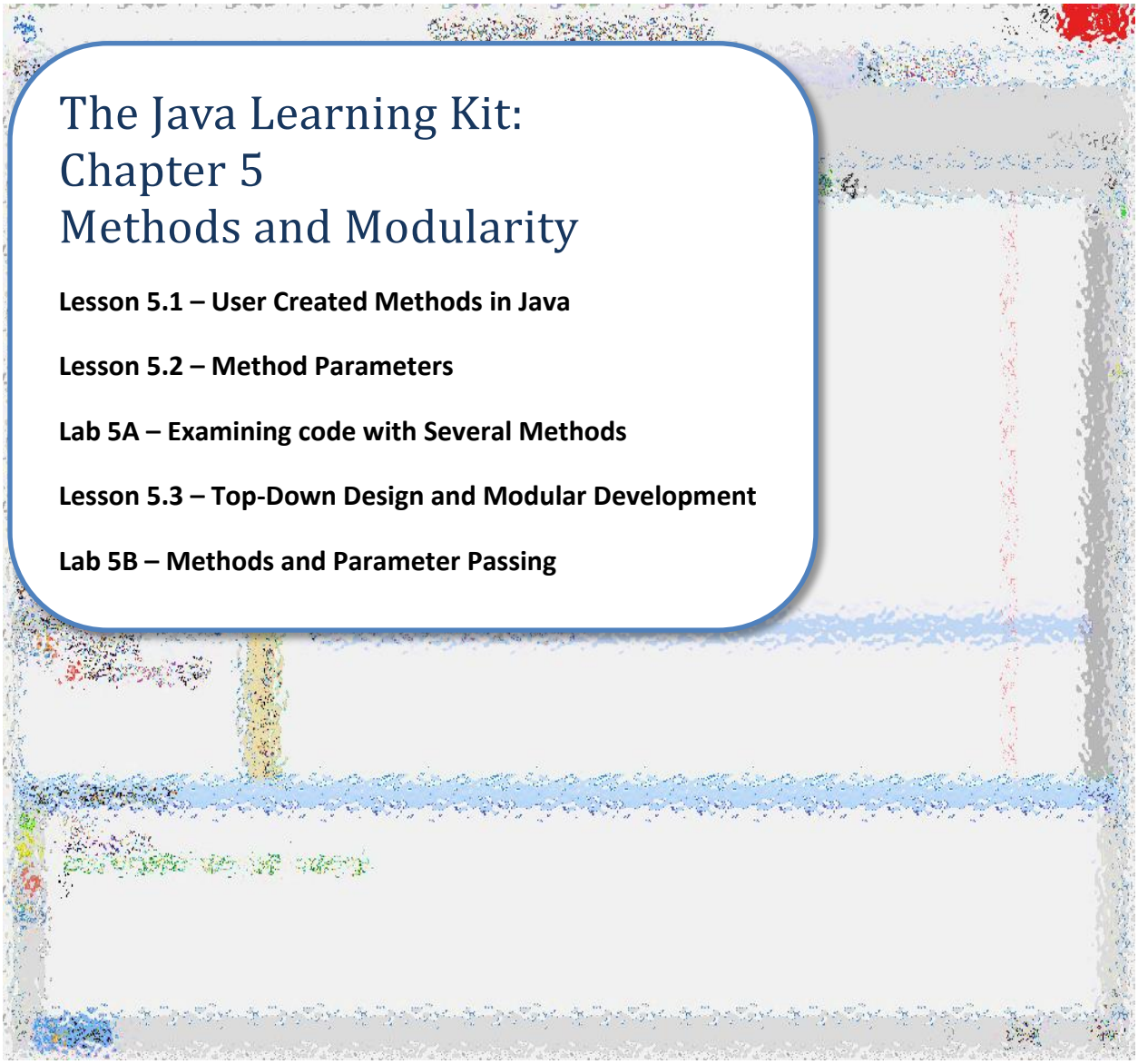


The Java Learning Kit: Chapter 5 Methods and Modularity

Lesson 5.1 – User Created Methods in Java

Lesson 5.2 – Method Parameters

Lab 5A – Examining code with Several Methods

Lesson 5.3 – Top-Down Design and Modular Development

Lab 5B – Methods and Parameter Passing

The Java Learning Kit: Chapter 5 – Methods and Modularity

Last edited January, 2015 by C. Herbert

This document is a chapter from a draft of the Java Learning Kit, written by Charles Herbert, with editorial input from Craig Nelson, Christopher Quinones, Matthew Staley, and Daphne Herbert. It is available free of charge for students in Computer Science courses at Community College of Philadelphia during the Spring 2015 semester.

Contents

The Java Learning Kit: Chapter 2
Reading, Writing, and Arithmetic

This chapter introduces user-created methods in Java, along with the accompanying techniques of modular development and top-down design, also known as structured decomposition or functional decomposition.

Parameter passing between methods and polymorphic method overloading are also discussed.

Chapter 5 Learning Outcomes

Upon completion of this chapter students should be able to:

- describe how to create methods in a Java class
- list and describe the six parts of a Java method, according to Oracle's *Java Tutorials.*
- describe how to pass values from one method to another and the difference between a formal parameter list and an actual parameter list.
- describe the concept of *top-down design* of an algorithm, also known as *structured decomposition* or *functional decomposition.*
- describe the concept of *modular development* and several advantages of developing software in smaller modules.
- create Java software with multiple methods in a class demonstrating good modular design and the proper use of parameter passing between methods.

Lesson 5.1 User Created Methods in Java

So far we have only created software with a single method, the main method in the single class in each Java application we created. We have invoked other methods from other classes, such as the Math class and the Scanner class, but we have only created one method at a time in our applications. Now we will begin to work with multiple methods of our own.

The code for each method in Java software is defined in a method declaration within a class. A **method declaration** defines a method, with a method header, followed by a block of code marked by braces. According to Oracle's Java tutorials, a method declaration has six components – the five parts of the method header and the method's block of code[1]. A Java **method header** includes method modifiers (such as *public* and *static)*, the return type of the method, the method name, a set of parentheses which contains the method's parameters, and an exception list. The parentheses are blank if there are no parameters. In general the parts of a method header look like this:

[*modifiers*] [*return type*] method name(*formal parameter list*) [*exception list*]

Here is an example. This method has no *exception list* after the parameter list. If you do see the word **throws** after a parameter list, then that is the beginning of the exception list, which is also known as a *thrown exception clause*. We will learn about exceptions later in the course.

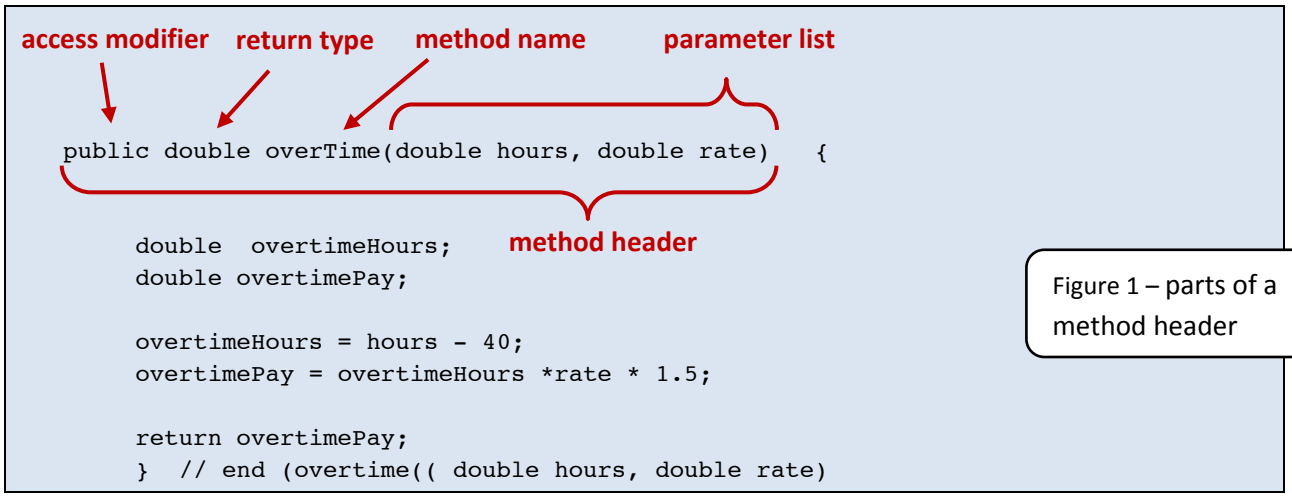


Figure 1 – parts of a method header

Method Modifiers

Method modifiers provide the compiler with information about the nature of the method. The most common method modifier is called the **access modifier**, which sets the access level for the method. Java methods may have public, private, or protected access.

- **Public access** means that the method may be invoked from any other software.
- **Private access** means the method may only be invoked from within the class which contains the method.

[1] See: http://docs.oracle.com/javase/tutorial/java/javaOO/methods.html

- **Protected access** is similar to private, but methods in subclasses of a class may invoke protected methods. We learn about subclasses later in the semester when we learn about inheritance.

Why Java Applications Need a *Public Static Void Main()* Method

Every java application must start somewhere.

A Java application can contain many methods in many different classes and can also import methods from classes that are not part of the application's own source code, but that are defined elsewhere.

At least one class in a Java application must have the same name as the application's software package. This class must contain a method named *main*, which will be the method that is executed when the software application runs. The *main* method will contain any calls or references to other methods used within the application.

A Java application's main method must be *public*, which means it is available to be run from outside of its own software package. This is necessary so that the operating system can start the application by running the main method. This means that the access modifier *public* must be included in a main method's header.

An application's main method is a class-level method, associated with the overall application's and not with an instance of a class, therefore its header must have the access modifier *static,* which identifies class-level methods, variables, and so on.

The main method for a java application does not return any values. Hence it must be marked with the access modifier *void* in the method's header, which indicates it does not return any user defined values. Actually, when a java application terminates it will send a code to the operating system to indicate whether or not the application terminated normally. However, this message passing happens automatically, and we do not need to be concerned about it, at least not for now.)

Putting all of this together, the modifiers *public*, *static* and *void* must come before the method name in the method header for a main method. For example, a payroll application whose application software package is named Payroll, must have a payroll class with the following method:

```
public static void main( String[] args)  {
    . . . method block of code follows  . . .
    } // end main()
```

The parameter for the main method is a String array of arguments, which is why main methods have the default parameter *String[] args.* (The terms *parameter* and *argument* mean almost the same thing, and are often used interchangeably.) Often this parameter goes unused, but it is there to enable operating system and the method to communicate with one another.

Another common method modifier is the static modifier. According to the JLS, a method that is declared static is called a *class method*. A **static method** is associated with the class and not with an instance of the class. It is invoked by using the class name, and not the instance name, as part of its qualified name.

The main methods in the software we have written so far have been declared *public static void main(…)*. We have also seen static methods in the Math class. Methods such as Math.sqrt() and Math.random() are class level methods that have been invoked using the name of the class – the Math class.

The Scanner class methods we have used were not class level methods. They were invoked using the name of an instance of the class. We had to declare a new instance of the Scanner class associated with a particular input stream before we could use methods such as nextLIne() in the example below. If the static modifier is missing, then the method is an instance method. If the static modifier is present, it is a class method.

Figure 2 – part of a *main()* method

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

String name;
Scanner kb = new Scanner(System.in);

// say hello to the user and ask for the user's name
 System.out.println("Hello, please enter your name: ");
 name = kb.nextLine();


. . .  // the method continues . . .
```

The static modifier identifies the main() method as a class-level method.

The *nextLine()* method is a instance method, invoked using the name of the instance of the Scanner class to which it applies. *kb*.

Return Type

The **return type** in a method header declares the data type of the value the method returns, or uses the keyword **void** to indicate that the method does not return a value. A method that does not return a value is often called a *void method.* A void method is invoked from another method by using the void method's name as if it is a new Java instruction.

A method that returns a value can called from anyplace in another Java method where a variable or literal of the same data type can be used. For example, the *Math.sqrt()* method returns a value of data type *double*. Therefore, it can be called from any place that a variable type *double* or a *double* literal can be used, such as in a math expression.

Methods that return a value, such as the *Math.sqrt()* method, are often called **inline methods**, because they can be invoked in a Java statement "*inline*", as shown on the left.

```
double result;
double num1;
result = 3 * num1 + 4;
result = 3 * Math.sqrt(10) + 4;
```

The *nextLine()* method that is invoked from the main() method in Figure 2, above, has a return type of String, so it can be used in a Java statement anyplace that a String value can be used. In this example, the value returned by the method is assigned to the String variable *name.*

Other method modifiers

There are several other modifiers, such as *abstract*, *native*, *strictfp*, and *synchronized*, that can be used in Java methods headers in addition to modifiers and return types discussed here. They are beyond what we need to use for now, but are listed and described online at:
http://docs.oracle.com/javase/tutorial/reflect/member/methodModifiers.html

Method Parameters

The list of variable declarations in parentheses following the name of the method is the formal parameter list. **Formal parameters** are local variables that will hold the values passed to the method when the method is invoked. They are declared in the method header by listing the data types and names of the variables. They may be used in the method just as any other local variables may be used. The scope of the variables declared in a formal parameter list is the body of the method in whose header the list appears.

In the overtime example shown again in Figure 3, the double variables hours and rate are formal parameters, which are used as local variables within the method. Values will be passed to initialize these variables when the method is invoked from another method. The parameters that are passed to a method are also called the arguments of a method.

```
/*************************************************************************/
/* The overtime method calculates the total overtime pay following the time
 * and a half for overtime rule. The result is to be added to the regular pay
 * for 40 hours.
 *
 * method parameters:
 * double hours -- hours worked this week – usually close to 40.0
 * double rate -- employee's pay rate  a decimal values usually less than 100
 *
 * The method returns the total overtime pay for hours above 40 as a double.
 */
   public double overTime( double hours, double rate) {
       double  overtimeHours;  // hours worked beyond 40
       double overtimePay;     // pay for overtime hours

       // calculate overtime hours and pay
       overtimeHours = hours - 40;
       overtimePay = overtimeHours *rate * 1.5;

       return overtimePay;
   }  // end overtime (double hours, double rate)
/*************************************************************************/
```

Figure 3 – a user-defined method

Invoking a Method and Parameter Passing

There are two ways a method may be invoked from another method:

- a method that returns a value may be used in an expression in a Java statement;
- a void method may be used in another method as if it is a new user-created instruction in the Java language.

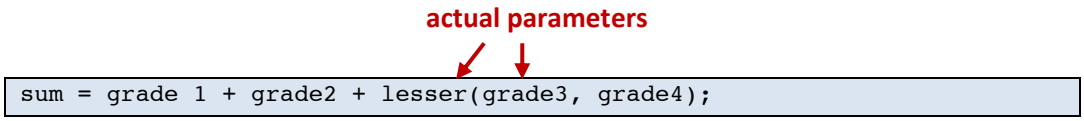
The following examples show how to invoke methods.

Example 1 - is a declaration for a method that returns a value. We can see from the method header that its return type is *int.* It also takes two integer parameters, *a* and *b.*

```
// this method returns the lesser of two integer values
public int lesser(  int a, int b)  {
      if ( b < a)
            return b;
      else
            return a;
      }  // end lesser()
```

Figure 4 – a method that returns an *int* value

The method may be used anyplace in a Java expression where an integer value can be used, such as in the following statement in another method:



In Java, parameters can be passed by value. **Parameter passing by value** means that the values specified in the actual parameter list are passed to the variables declared in the formal parameter list. When a method is invoked, the data types of the actual parameters must match those of the formal parameters and the actual parameters must be in the same order as the formal parameters.

In this case, the values of the variables *grade3* and *grade4* are the actual parameters for this invocation of the method. They are passed to the variables *a* and *b* in the method *lesser()*. The values of the variables *grade3* and *grade4* are used to initialize the variables *a* and *b* in the method *lesser().*

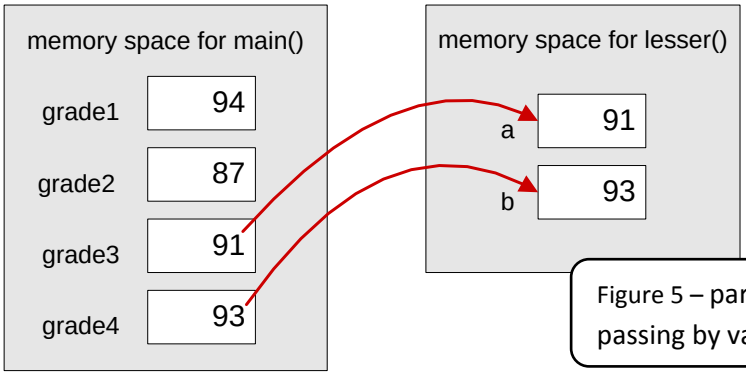


Figure 5 – parameter passing by value

If *grade3* = 91 and *grade4* = 93, then 91 is passed to *a* and 93 is passed to *b*. There is no connection between the variables in the two methods – the values from one are simply passed to the other.

Example 2 - a value returning method named square

```
// this method returns the square of a double value
public double square( double x) {
      return x * x;
      }  // end square()
```

Figure 6 – a method that returns a double value

The square() method is invoked in each of the following statements:

```
dist = Math.sqrt( square(dx) + square(dy) );
area = square(s);
area = square(2.5);
```

The *square()* method is called four times, twice in the first statement, once in the second statement., and once in the third statement. Notice that the third statement uses the value *2.5* as an actual parameter – no variable is used. This is perfectly acceptable when *passing by value*, as Java does.

In the first statement, the value of *dx* is passed to *square()* and the return value is used in place of the method call *square(dx)*, then the value of *dy* is passed to *square()* and the return value is used in place of the method call *square(dy).*

In the second statement, the value of s is passed to the *square()* method and the formal parameter *x* is initialized to the value of *s*. *Area* will be equal to s^2.

In the third statement, the value *2.5* is passed to square and the formal parameter *x* is initialized to *2.5*. Area will be equal to 2.5^2, which is *6.25*.

****

Example 3 - a void method

```
public void printCheck(String name, double netPay) {
    System.out.println("Pay to the order of " + name);
    System.out.printf("%8.2f", netPay )
    } // end printCheck()
```

Figure 7 – a void method – used "inline" as if it is a new Java instruction

A void method is used as if it is a new instruction in the Java language:

```
printCheck(name, netPay);
printCheck("Mary Jones", 371.28);
printCheck( (fname + " " + lname), net);
```

In each case, the program will invoke printCheck(), passing the values to the method. The method uses the data to print the checks, but no value is returned to the calling method.

The parameter passing works the same with void methods as with value returning methods: the values in the actual parameter list are used to initialize the variables in the formal parameter list. In the first statement above, the values of the variables *name* and *netPay* in the calling method are used to initialize *empName* and *netPay* in the printCheck() method. Even though the two variables *netPay* have the same name, they are still different variables not connected to one another in any way. The value of one is simply used to initialize the other.

In the second *printCheck()* statements above, literal values are specified. These values will be used to initialize *name* and *netPay* when *printCheck()* is invoked.

Method Documentation

The version of the overtime method shown in Figure 3 is reasonably well-documented. Documentation is an important part of programming. Method documentation is an important part of object-oriented programming.

Information in a comment describing each method should include the method's:

- entry conditions
- what the method does and how the method functions
- exit conditions

A method's **entry conditions** include information about the variables in the formal parameter list – such as the range of values for each parameter – along with any other special conditions that must be set before the method runs. In the case of the overTime method in Figure 3, the expected values of the variables hours and rate are listed, but no special conditions are mentioned.

The method documentation should briefly describe what the method does and how the method functions. It should tell the reader if any new or unusual techniques are used in the method. If there are several different way to do something, then the documentation should be clear about which way is used in this method. The documentation for the *overTime()* method in Figure 3 makes it clear that the "time and a half for overtime" rule was applied, and that the method returns the total pay for all overtime hours, not just the added half.

A method's **exit conditions** should describe what value, if any, a method returns and anything else the method does that has an effect outside of the method. Does it open a new I/O stream? Does it turn a hardware device on or off? If it does anything with an effect that will be present after the method runs, then the exit conditions should say so. In the case of the overtime method in Figure 3, the only exit condition is that it returns a value. The data type and meaning of the value returned should be clearly described.

You should also do something with the documentation to clearly separate one method from another in a class. Programming styles differ, so there are many ways to do this. In the case of the overtime method, a line of stars and then a blank line are immediately placed at the end of the method. This makes it easier for a reader to distinguish between methods.

Remember, the documentation will be stripped from the code when the software is compiled. It's there to make it easier for people to read and understand the code, including you as the author of the code who may need to come back to a method months or years later and will need to understand how it works. All documentation should be clear and reasonably concise, but ease of understanding is the most important factor. It is worth the time to make sure that readers can understand your code.

CheckPoint 5.1

1. What are the six parts of a method declaration in Java, according to Oracle's *Java Tutorials*?
2. What do the method modifiers *public*, *private*, and *static* each mean in a method declaration?
3. What is a void method and how is it invoked form another method?
4. How is a value returning method invoked from another method?
5. What is a formal parameter list and how does it correspond to an actual parameter list?

Lab 5A – Examining code with Several Methods

The following source code is in the NetBeans project *PayMethods*, included in the file payMethods.zip in the files for Week 5 in Canvas. It demonstrates the use of a value returning method and a void method.

Your task is to download the file, unzip the NetBeans project, then open it in NetBeans and examine the code. You should be able to see how the a value returning method and a voids method are invoked from main, and how parameters are passed from the main method to calculateGross() and to payrollReport().

Notice that calculateGross() and payrollReport() are both static methods. This is because main() is a static method, and static methods cannot call instance methods.

You should run the program to see how it works, then perhaps try changing some of the actual parameters to specific values to see how that works.

Notice the modularity of the program – we could change the print report formats, for example, without messing with the *main()* method. We could also modify the way overtime is calculated without changing any other methods.

```
package paymethods;
import java.util.Scanner;

/* CSCI 111 Fall 2013
 * Sample program - payroll methods
 * This program has an example of a value returning method and a void method
 */

public class PayMethods {

    public static void main(String[] args) {
    String fname;       // employee's first name
    String lname;       // employee's last name
    double hours;       // hours worked in the week
    double rate;        // hourly pay rate
    double gross;       // gross pay (hours * rate) + overtime

    // set up input stream from the keyboard
    Scanner keyboard = new Scanner(System.in);

    // get employee's first name
    System.out.print("Please enter the employee's first name: " );
    fname = keyboard.next();

    // get employee's last name
    System.out.print("Please enter the employee's last name: " );
    lname = keyboard.next();

    // get employee's hours
    System.out.print("Please enter the hours for " + fname +" " + lname+ ": ");
    hours = keyboard.nextDouble();

    // get employee's rate
    System.out.print("Please enter the pay rate for " + fname +" " + lname+ ": ");
    rate  = keyboard.nextDouble();
```

Figure 8 – a payroll program composed of several methods

```
    // call method to calculate gross
    // This value returning method is used inline, gross is set to the value it returns
    gross = calculateGross(hours, rate);

    // call method to print payroll report
    // this void method is used as if it is a new instruction in Java
    // the actual parameters must correspond with the method's formal parameter list
    payrollReport(fname, lname,  hours, rate, gross );

    } // end main()
    /*************************************************************************/

    /* This method calculates gross pay using the time and a half for overtime rule.
     * it has two parameters,
     *      the hours worked during the week    double hrs
     *      the pay rate                        double rt
     *
     * it returns the gross pay
     */
    public static double  calculateGross(double hrs, double rt ) {
        double gr;  // gross pay
        double ot;  // overtime pay

        gr = hrs * rt;

        if (hrs > 40)
            ot = (hrs-40) * .5 * rt;    // overtime bonus
        else
            ot = 0;

        gr = gr + ot;
        return gr;

    } // end calculateGross
/*************************************************************************/

    /* this method prints a payroll report
     * it takes four parameters,
     *      employeee's first name              String fname
     *      employee's last name                String lname
     *      the hours worked during the week    double hrs
     *      the pay rate                        double rate
     */
      public static void payrollReport(String fname, String lname,
            double hrs, double rt, double gross ) {

             // print the payroll report
             System.out.printf("%n%-12s%-12s%8s%8s%9s%n", "first", "last",
                    "Hours", "Rate", "Gross");

             System.out.printf("%-12s%-12s%8.2f%8.2f%,9.2f%n", fname, lname,
                    hrs,rt, gross);

      } // end payrollReport()

} // end class
```

Lesson 5.2 Top-Down Design and Modular Development

It's hard to solve big problems.

It's easier to solve small problems than it is to solve big ones.

Computer programmers use a divide and conquer approach to problem solving:

- a problem is broken into parts
- those parts are solved individually
- the smaller solutions are assembled into a big solution

This process is a combination of techniques known as ***top-down design*** and ***modular development***.

Top-down design is the process of designing a solution to a problem by systematically breaking a problem into smaller, more manageable parts.

First, start with a clear statement of the problem or concept – a single big idea. Next, break it down into several parts. If any of those parts can be further broken down, then the process continues until you have a collection of parts that each do one thing.

The final design might look something like this **organizational chart**, showing the overall structure of separate units that form a single complex entity.

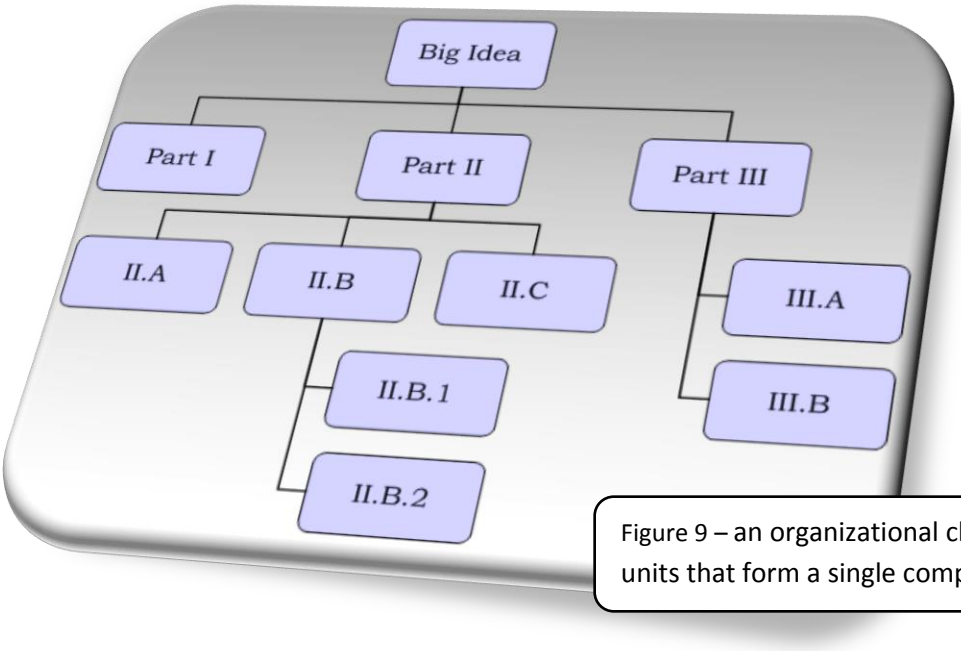


Figure 9 – an organizational chart showing units that form a single complex entity

An organizational chart is like an upside down tree, with nodes representing each process. The **leaf nodes** are those at the end of each branch of the tree. The leaf nodes represent modules that need to be developed and then recombined to create the overall solution to the original problem.

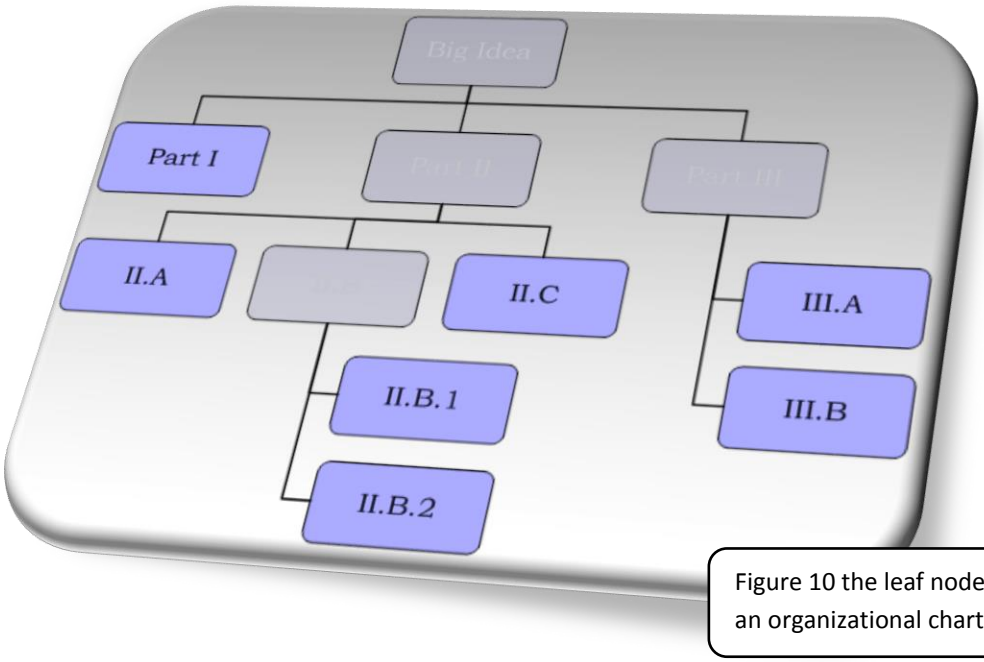


Figure 10 the leaf nodes of an organizational chart

Top-down design leads to ***modular development***.**Modular development** is the process of developing software modules individually, then combining the modules to form a solution to an overall problem.

Modular development facilitates production of computer software because it:

… makes a large project more manageable.

Smaller and less complex tasks are easier to understand than larger ones and are less demanding of resources.

… is faster for large projects.

Different people can work on different modules, and then put their work together. This means that different modules can be developed at the same time, which speeds up the overall project.

… leads to a higher quality product.

Programmers with knowledge and skills in a specific area, such as graphics, accounting, or data communications, can be assigned to the parts of the project that require those skills.

…makes it easier to find and correct errors.

Often, the hardest part of correcting an error in computer software is finding out exactly what is causing the error. Modular development makes it easier to isolate the part of the software that is causing trouble.

… increases the reusability of solutions.

Solutions to smaller problems are more likely to be useful elsewhere than solutions to bigger problems. They are more likely to be ***reusable code***. **Reusable code** is code that can be written once, then called upon again in similar situations. It makes programming easier because you only need to develop the solution to a problem once; then you can call up that code whenever you need it. Modules developed as part of one project, can be reused later as parts of other projects, modified if necessary to fit new situations. Over time, libraries of software modules for different tasks can be created. Libraries of objects can be created using object-oriented programming languages.

Most computer systems are filled with layers of short programming modules that are constantly reused in different situations. Our challenge as programmers is to decide what the modules should be. Each module should carry out one clearly defined process. It should be easy to understand what the module does. Each module should form a single complete process that makes sense all by itself.

Top-down development is used to figure out what the modules are needed in a software development project and how they should be organized. Modular development involves building those modules as separate methods, then combining them to form a complete software solution for the project.

CheckPoint 5.2

1. What are Java's nine built-in data types and how do they fit into four major categories?
2. Which Java built-in data type is not a primitive data type?
3. How do each of Java's integer data types differ from one another?
4. What is a literal and how are literals represented for different built-in data types?
5. Describe the formats used to store floating point data in memory.

Lab 5B – Methods and Parameter Passing

In this example we will develop a modular solution for a problem that is similar to the homework assignment from chapter 3.

We wish to input the x and y coordinates for two points in the Cartesian plane, and find the distance between the two, then print a report telling the user the distance between the two points and the quadrant that each point is in.

The distance between two points whose coordinates are (x_1,y_1) and (x_2, y_2) is $\sqrt{\Delta x^2 + \Delta y^2}$ where Δx is the difference between the two x coordinates (x_1- x_2)and Δy is the difference between the y coordinates (y_1- y_2).

The example on the right also shows us that the quadrants of a Cartesian plane are numbered I through IV, with the following properties:

- If both x and y are non-negative, the point is in Quadrant I.
- If x is negative and y is non-negative, the point is in Quadrant II
- If x and y are both negative, the point is in Quadrant III.
- If x is non-negative and y is negative, the point is in Quadrant IV.

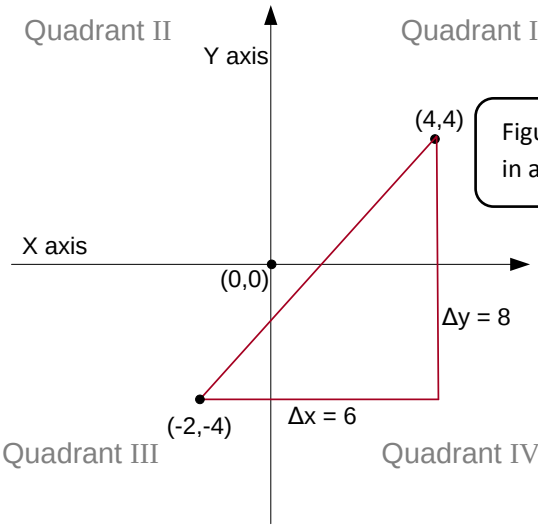


Figure 11 – two points in a Cartesian plane

Once we understand the specifications, we can make an outline of what our software should do:

1. get the coordinates of two points: (x1, y1) and (x2, y2)
2. calculate the distance between the two points:
   - deltaX =x1-x2; deltaY = y1-y2;
   - dist = Math.hypot(deltaX, deltaY);
3. determine which quadrant (x1, y1) is in:
   - If (x >=0) && y>=0) → Quadrant I
   - If (x <0) && y>=0) → Quadrant II
   - If (x <0) && y<0) → Quadrant III
   - If (x >=0) && y<0) → Quadrant IV
4. determine which quadrant (x2, y2) is in:
   - If (x >=0) && y>=0) → Quadrant I
   - If (x <0) && y>=0) → Quadrant II
   - If (x <0) && y<0) → Quadrant III
   - If (x >=0) && y<0) → Quadrant IV
5. print results

The similarity of steps 3 and 4 indicate that this would be good place for reusable code – one method that can be used twice, but each time with different values passed to the method. Step 2 is also a good place for a method, with the details of the calculation in the method.

Step 1 might seem like a good place for a method – we get four numbers from the user, each in the same manner. However, remember that methods can only return one value, so we really don't gain much by making this a separate method. It is a judgment call and a matter of programming style – it could be done with four calls to one method, but in this case we will use four different sets of statements in the main method.

Here in the design of our software, listing descriptions of the methods with return types and formal parameters:

Void main() method

- Get input
- Call method to calculate distance
- Call method to determine quadrant of point 1
- Call method to determine quadrant of point 2
- Print results

Double CalcDistance(double x1, double y1, double x2, double y2) method

- Calculate deltaX
- Calculate deltaY
- Calculate and return distance

String findQuadrant(double x, double y)

- if (X >=0) && (Y>=0) Q = "Quadrant I"
- if (X <0) && (Y>=0) Q ="Quadrant II"
- if (X <0) && (Y<0) Q ="Quadrant III"
- if (X >=0) && (Y<0) Q ="Quadrant IV"
- return Q

The code in Figure 12 starting on the next page was developed from these specifications.

```
/*
 * program to calculate the distance between two points
 * CSCI 111 Fall 2103
 * last edited Sept 24, 2013 by C. Herbert
 */
package twopoints;
import java.util.Scanner;

public class TwoPoints {

    // The main method gets x and y for two points
    // Calls methods to calculate the distance between the two
    // and to determine the quadrant of each point
    // then outputs the result
    public static void main(String[] args) {
         //declare variables

         double x1,y1,x2,y2;    // coordinates of (x1,y1) and (x2,y2)
         double dist;           // distance between (x1,y1) and (x2,y2)
         String Q1, Q2;         // quadrant containing (x1,y1) and (x2,y2)

         // set up instacne of Scanner for input
         Scanner kb = new Scanner(System.in);

         // get input in prompt & capture pairs
         System.out.print("Enter the x coordinate of point 1 :");
         x1 = kb.nextDouble();

         System.out.print("Enter the y coordinate of point 1 :");
         y1 = kb.nextDouble();

         System.out.print("Enter the x coordinate of point 2 :");
         x2 = kb.nextDouble();

         System.out.print("Enter the y coordinate of point 2 :");
         y2 = kb.nextDouble();

         // call calcDistance()
         dist = calcDistance( x1, y1, x2, y2);

         // call findQuadrant(point1)
         Q1 = findQuadrant(x1, y1);

         // call findQuadrant(point2)
         Q2 = findQuadrant(x2, y2);

         // output results
         System.out.println("\nPoint 1 is ("+ x1 + "," + y1 + ")");
         System.out.println("Point 2 is ("+ x2 + "," + y2 + ")\n");

         System.out.printf("The distance between the two points is: %-8.2f%n", dist );
         System.out.println("Point 1 is in " + Q1);
         System.out.println("Point 2 is in " + Q2);

     } // end main()
/*********************************************************************/
```

Figure 12 – a program organized as 3 methods

```
    // this method calculates and returns the distance between two points
    public static double calcDistance(double x1,double y1,double x2,double y2)  {
        double deltaX;     // difference in x coordinates
        double deltaY;     // difference in x coordinates
        double dist;       // distance between (x1,y1) and (x2,y2)

        deltaX = (x1-x2);
        deltaY = (y1-y2);

        dist = Math.hypot(deltaX, deltaY);
        return dist;

        /* this whole method could be one line:
         * return Math.hypot( (x1-x2), (y1-y2) );
         */
    } // end calcDistance()  {
/*************************************************************************/
    // this method returns the quadrant that contains the point

    public static String findQuadrant(double x,double y)  {
        String Q;      //quadrant message

       if ( (x >= 0.0) && (y >= 0.0) )
           Q = "Quadrant I";
       else if ( (x < 0.0 ) && ( y >= 0.0) )
           Q = "Quadrant II";
       else if ( (x < 0) && (y < 0.0) )
           Q ="Quadrant III";
       else // ( (x >= 0) && (y < 0.0) )
           Q = "Quadrant IV";

       return Q;
    } // end findQuadrant

} // end class
```

Here is sample output from the program:

```
Enter the x coordinate of point 1 :3
Enter the y coordinate of point 1 :4
Enter the x coordinate of point 2 :-3
Enter the y coordinate of point 2 :-4

Point 1 is (3.0,4.0)
Point 2 is (-3.0,-4.0)

The distance between the two points is: 10.00
Point 1 is in Quadrant I
Point 2 is in Quadrant III
```

The code in Figure 13 on the next page contains an alternative version of the same program as Figure 12. To the user, the two programs look the same. They have identical input and output. They are in the Week 5 Canvas files. These two Java applications are the topic of the Week 5 class discussion.

```
/* distance between two points - alternative programming style
 * CSCI 111 Spring 2015  last edited Sept 24, 2013 by C. Herbert
 */
package twopointsb;
import java.util.Scanner;

public class TwoPointsB {

    // The main method calls a methods to calculate the distance between
    // two points, then prints that distance and the quadrant of each point
    public static void main(String[] args)    {

         double x1,y1,x2,y2;
         double dist;

         // input coordinates
        x1 = inCor("x", "1");
         y1 = inCor("y", "1");
         x2 = inCor("x", "2");
         y2 = inCor("y", "2");

         dist = Math.hypot( (x1-x2), (y1-y2) );

         System.out.println("\nPoint 1 is ("+ x1 + "," + y1 + ")");
         System.out.println("Point 2 is ("+ x2 + "," + y2 + ")\n");
         System.out.printf("The distance between the two points is: %-8.2f%n", dist );
         printQuadrant("1", x1 , y1);
         printQuadrant("2", x2 , y2);
     } // end main()
/*************************************************************************/

       // this method gets as input a single coordinate of a point
      public static double inCor(String axis, String point) {

      // set up instance of Scanner for input
      Scanner kb = new Scanner(System.in);
        // get coordinate with prompt
      System.out.print("Enter the " + axis + " coordinate of point " + point + " :");
      return kb.nextDouble();
     } // end inCor()
/*************************************************************************/

     // this method prints the quadrant a point is in
     public static void printQuadrant(String P, double x,double y) {

        if ( (x >= 0.0) && (y >= 0.00) )
            System.out.println( "Point "+ P + " is in Quadrant I");
        else if ( (x < 0.0 ) && ( y >= 0) )
            System.out.println( "Point "+ P + " is in Quadrant II");
        else if ( (x < 0) && (y < 0) )
            System.out.println( "Point "+ P + " is in Quadrant III");
        else // ( (x >= 0) && (y < 0) )
            System.out.println( "Point "+ P + " is in Quadrant IV");
     } // end printQuadrant
} // end class
```

Figure 13 –Figure 12's program reorganized.

Key Terms

access modifier, 4

entry conditions, 10

exit conditions, 10

formal parameters, 7

inline methods, 6

leaf nodes, 15

method declaration, 4

method header, 4

method modifiers, 4

modular development, 16

organizational chart, 14

parameter passing by value, 8

private access, 4

protected access, 4

public access, 4

return type, 6

reusable code, 16

static method, 6

throws, 4

top-down design, 14

void, 6

Chapter Questions

1. So far this semester we have only created one method at a time in our applications, but what are some examples of methods from other classes that we have invoked in our source code?
2. What is included in a method declaration?
3. What are the parts of a method header?
4. What is in the parentheses that follow the name of the method in a method header? What does it mean if the parentheses are blank?
5. What does it mean if the word *throws* follows a method header?
6. Where does the return type belong in a method header?
7. What are the three access modifiers used in Java? What does the *private* access modifier do?
8. When do we need a main method in a class? What modifiers should be used with this method?
9. What does the JLS tell us about a method with the modifier static? How is a static method invoked?
10. What methods that we have previously used were static methods? What methods that we have previously used were associated with an instance of a class?
11. What do we call a method that does not return as value? What return type does it have?
12. Where can we use a method that returns a double value?
13. What is a method that can be invoked in a Java expression often called?
14. What is the difference between the formal parameter list and the actual parameters? Which of the two declares new variables for a method? Which contains values used to initialize variables?

15. What should be included in a comment describing each method? What happens to this documentation when a method is compiled?
16. What should be described as part of a method's entry conditions? What should be described as part of a method's exit conditions?
17. In what two ways can a method be invoked from another method? How is a value returning method invoked? How is a void method invoked?
18. How are parameters passed in Java? How must the values of the actual parameters be organized when a method is invoked?
19. When breaking a method down into parts using top-down design, which nodes on an organizational chart need to be developed and then recombined to create the overall solution to the original problem?
20. What are five of the ways in which modular development facilitates production of computer software?

Chapter Exercises

1. Match the actual parameters on the left with the correct formal parameter list on the right.

(3,4) (2.0,3.0) ("Jones", 97.5) ("J00123453", "B") (42, 10.50)	(double hours, double rate) (String name, double pay) (short people, double PoundsOfCake) (String Jnumber, String grade) (int length, int width)

2. Multiplication Table

   A set of nested *for* loops can be used to print a multiplication table using *System.out.printf* statements.

	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	45
6	6	12	18	24	30	36	42	48	54
7	7	14	21	28	35	42	49	56	63
8	8	16	24	32	40	48	56	64	72
9	9	18	27	36	45	54	63	72	81

   Write a program to print such a multiplication table, using a method to print each row, which is invoked from the method that prints the overall multiplication table.

3. Printing Patterns

   Nested *for* loops can print patterns of stars, such as the example below:

```
for(i=1; i<=5; i++)
{
    for(j=1; j<=10; j++)
    {

System.out.print("*");
    } // end for j
    System.out.println();
 }  // end for i
```

Output from this program:

```
**********
**********
**********
**********
**********
```

   Write a program with a main method which asks the user for the number of rows and for the number of columns, then calls a method which uses that information to print a rectangle of stars.

4. Celsius to Fahrenheit Table

   Write a program to print a Celsius to Fahrenheit conversion table that invokes a method to calculate Fahrenheit temperature using formula is F = $\left(\frac{9.0}{5.0}\ C\right)$ + 32.0. The main method should have a loop to print the table, calling the conversion method from within the loop.

   The table should include integer Celsius degrees from 0 to 40. You program does not need to use integers, but the table should display the Celsius temperature, then the Fahrenheit accurate to one-tenth of a degree. The first few entries from the table are shown below:

Celsius	Fahrenheit
0	32.0
1	33.8
2	35.6
3	37.4

5. We wish to write a modular program that can be used to calculate monthly payments on a car loan. The formula is: *payment* = $\frac{montly\ rate * amount}{1-(1+montly\ rate)^{-months}}$

   The program will be composed of several methods – a main method, a monthly interest rate method, a monthly payment method, and an output method.

   The main method should:

   1. ask the user for
      - the loan amount
      - the annual interest rate *(as a decimal, 7.5% is 0.075)*
      - the number of months
   2. call a method to calculate and return the monthly payment
      (Note: The formula uses monthly rate. Annual rate is input. monthly rate = annual rate/12)
   3. call a method to print a loan statement showing the amount borrowed, the annual interest rate, the number of months, and the monthly payment

6. International Gymnastics Scoring

We wish to write a program to quickly calculate gymnastics scores for an international competition. Six judges are judging the competition, numbered 1 through 6 for anonymity. The program should have a loop that:

1. calls a method asking for the score from a single judge, The method should print the judge number and score and return the judges score
2. adds the returned score to a total score.

After the loop is done, your program should call a method that calculates and displays the average score.

Scores are in the range 0 to 10 with one decimal place, such as 8.5. The average should be displayed with two decimal places.

Here is a sample of the output:

```
Score for Judge 1:  8.5
Score for Judge 2:  8.7
Score for Judge 3:  8.4
Score for Judge 4:  9.1
Score for Judge 5:  8.9
Score for Judge 6:  9.0
The average score is:  8.77
```

7. Test for Divisibility

If the remainder from dividing A by B is zero, then A is a multiple of B (A is evenly divisible by B).
For example, the remainder from dividing 6, 9, or 12 by 3 is 0, which means 6, 9, and 12 are all multiples of 3.

Write a program with a loop to to print the numbers from 1 to 25, and within the loop, invokes a method to see if the number is a multiple of 2, a method to see if the number is a multiple of 3, and a method to see if the number is a multiple of 5. The methods should each print a message if the number is a multiple of the value for which they are testing.

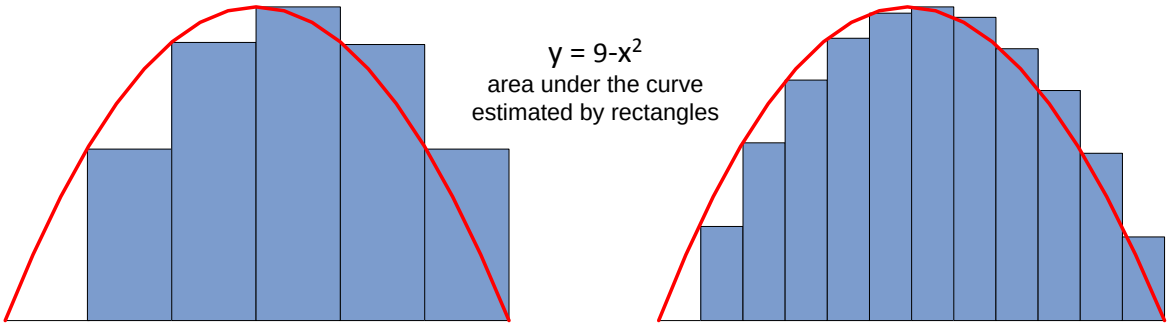
Here is a sample of part of the output:

```
1
2       multiple of 2
3       multiple of 3
4       multiple of 2
5       multiple of 5
6       multiple of 2     multiple of 3
7
8       multiple of 2
9       multiple of 3
10      multiple of 2     multiple of 5
```

8. Estimate of a Definite Integral

Write the program to estimate a definite integral with rectangles, with a loop that calls a method to calculate the height and area of each rectangle.

A loop in a computer program can be used to approximate the value of a definite integral. We can break the integral into rectangles, find the area of each rectangle then add the areas together to get the total area, as shown in the accompanying diagram. In practice, we could make the width small enough to achieved the desired accuracy in estimating the area under the curve.



As the width of the rectangles gets progressively smaller, their total area gets closer to the total area under the curve. This is an old method that predates Calculus. Both Leibnitz and Newton used this method, eventually developing calculus from the concept of infinitesimals, which, in the case of our rectangles, would amount to asking, what would the total area be if we had very many rectangles with very small width?

This was tedious by hand, but we now have computers. We can set the width of the rectangle. The height of the rectangle is the y value at point x. So for example, if we need to solve:

$$\int_{+3}^{-3} 9 - x^2 \, dx$$

The starting point is -3, the ending point is +3, and the height of the rectangle at each value of x in between the two is $9 - x^2$; Y at each x is the is the height of the rectangle at that x. We can use a width of 0.001, and write a loop like this:

```
for x = -3 to +3  step 0.0001   // x is -3.00, then -2.9, then -2.8, etc.
  {
    y = 9 – x² ;              // the height is the y value at each x
    area = y * .01;           // the width is the increment, in this case 0.1
    totalArea = totalArea + area;
  }
```

In this version of the program, the statements in the box above should be in a separate method, invoked from within the loop. When the loop is finished, totalArea is the total area of the rectangles, which is an approximation of the area under the curve.

Your lab report should explain what you did, and the result of your work.

9. Supermarket Checkout

   Design a program for a terminal in a checkout line at a supermarket. What are the tasks that are included as part of checking out at a register? We must consider things like the cost of each item, weighing produce and looking up produce codes, the total cost of the order, bonus card discounts, and sales tax on taxable items only. You do not need to write the program, but you should design the methods using top down development and modular design. Submit a description of the methods needed for the program, a brief description of what each method does, and a description of how the methods are related to one another in terms of parameter passing and return types.

10. We wish to write a program to draw the scene below. We can create methods to draw primitive shapes, as follows:
    circle() takes x and y coordinates of center point, the radius, and the color
    triangle() takes x and y coordinates of each of three endpoints and the color
    rectangle() takes x and y coordinates of each of four corner points and the color

    Create a modular design for software to draw the scene using the methods above. The software should be layered. For example, a *house()* method should use the primitive graphics methods to draw the house.

   Your design document should list and briefly describe each method, including and what it does, parameters, and any methods the method calls. Your list should start with the main method.

— End of Chapter 5 —