Chapter 8
Methods II

In Chapter 5 we introduced methods. In this chapter we take a much more in depth look at methods and how they are used to deal with complexity in programming.

Methods provide a powerful tool for dealing with the complexity of software development. They offer

- program modularization
- code reuse
- code simplification
- improved readability
- less source code
- easier debugging
- easier maintenance
- a step towards software libraries

Remember the Rule for Argument Passing—when a method call contains as argument, the content of the argument is copied and this copy is assigned to the parameter in the method definition.

8.1 Methods and Primitive Data Types

As a consequence of the Rule for Argument Passing primitive data types are passed by value and so changes to parameters inside a method do not change the arguments outside the method.

So far we've looked at void methods, which don't return a value. In this chapter we'll consider other varieties of methods which give us a lot of convenience and power, methods that return a value.

8.1.1 Numeric Methods and Primitive Data Types

A method that returns a single numeric value is a numeric method. We've been using numeric methods for quite a while.

```java
double s = Math.sqrt(38.67);
int xLoc = (int) circleOne.getX();
int v = readInt("Enter score");
```
A method that returns an `int` is called an *int method*. A method that returns a `double` is a *double method*.

Creating our own numeric methods is easy. Let's take a look at a program that calculates and displays the area and circumference of a circle given its radius \( r \).

The math we need for these calculations is
- area = \( \pi r^2 \)
- circumference = \( 2\pi r \)

### CircleCalcs

```java
//CircleCalcs.java
import acm.program.*;

public class CircleCalcs extends ConsoleProgram {
    public void run()
    {
        double radius, area;
        radius = readDouble("Radius? ");
        area = calcArea(radius);
        println("Area: " + area);
        println("Circumference: " + calcCircumference(radius));
    } //run

    public double calcArea(double r)
    {
        return Math.PI * Math.pow(r, 2);
    } //calcArea

    public double calcCircumference(double r)
    {
        return 2*Math.PI*r;
    } //calcCircumference
}

//CircleCalcs
```

Note that numeric methods don't have to be one liners. Our examples are because the mathematics required is very simple, but more complex math will require more code. For example, the familiar formula for calculating one root of a quadratic equation...
might be used to create this method.

```java
public double calcRoot1 (double a, double b, double c) {
    double numerator = -b + Math.sqrt(Math.pow(b, 2) - 4*a*c);
    double denominator = 2*a;
    return numerator/denominator;
} //calcRoot1
```

Of course we could write a similar `calcRoot2()` method that performs the calculation with – in front of the . But with a little cleverness we can write one method to do both.

```java
public double calcRoot(double a, double b, double c, double plusMinus) {
    double numerator = -b + plusMinus*Math.sqrt(Math.pow(b, 2) - 4*a*c);
    double denominator = 2*a;
    return numerator/denominator;
} //calcRoot
```

calcRoot( ) can be used to calculate the first root with `calcRoot(a, b, c, +1)` and the second root with `calcRoot(a, b, c, -1)` by using the last argument to set the calculation to use either addition or subtraction.

The returned type (`double` in the examples above) should match with the actual type being returned. The method

```java
public int thirdPower(double p) {
    return p * p * p;
} //thirdPower
```

has a problem. The compiler tells us there is a possible loss of precision because `p*p*p` is a `double` result (because `p` is a `double`) and our method is returning an `int`. A `double` takes more memory than an `int`, so we’ve got to allow for it. This method should return a `double`.

### 8.1.2 boolean Methods and Primitive Data Types

Often we need a method to test a value or an object to see if it meets some criteria, returning `true` if the criteria is met and `false` if it is not. Such a method is known as a **boolean method** and also as a **predicate method**.
We've already seen many boolean methods in the Character class—\texttt{isDigit()} \texttt{isLetter()} and so on. For practice, let's write our own \texttt{isLetter} method and look at a sample use.

```java
public boolean isLetter(char ch)
{
    if (('a' <= ch && ch <= 'z') || ('A' <= ch && ch <= 'Z'))
    {
        return true;
    } else
    {
        return false;
    }
} //isLetter
```

```java
if (isLetter(sampleChar) == true)
{
    do something
}
else
{
    do something else
}
```

Here's a boolean method that tests a numeric value to see if it's in a specific range.

```java
public boolean isInRange(int value, int minValue, int maxValue)
{
    if ((minValue <= value) && (value <= maxValue))
    {
        return true;
    } else
    {
        return false;
    }
} //isInRange
```

```java
if (isInRange(v, 10, 40) == true) {
    do something
} else {
    do something else
}
```

boolean methods are often named beginning with a verb such as ‘is’ or ‘has’ that indicates a state or condition, as in \texttt{isLetter()}, \texttt{isInRange()}, \texttt{containsX()}, \texttt{hasFinished()}, etc.

### 8.1.3 An Example Program

Now let's look at an example program that calculates the real roots of a quadratic

\[ ax^2 + bx + c = 0 \]

where a, b and c are real numbers.

The algorithm is

\textit{input a, b and c}
if the quadratic has real roots
{
    calculate the two roots
    display the roots
}
else
{
    display no real roots message
}

and here's a program to implement it.

QuadraticRoots
//QuadraticRoots.java
import acm.program.*;
public class QuadraticRoots extends ConsoleProgram
{
    public void run()
    {
        double a, b, c;
        double x1, x2;
        boolean exit = false;
        String answerLine;
        char answer;
        while(exit == false)
        {
            a = readDouble("First coefficient? ");
            b = readDouble("Second? ");
            c = readDouble("Third? ");

            if (hasRealRoots(a, b, c) == true)
            {
                x1 = calcRoot(a, b, c);
                x2 = calcRoot2(a, b, c);
                println("x1: "+ x1 + " x2: "+ x2);
            }
            else
            {
                println("No real roots.");
            }

            answerLine = readLine("Again? y/n ");
            answer = answerLine.charAt(0);
            if (answer == 'N' || answer == 'n')
            {
                exit = true;
            }
        } //while
        println("nExiting QuadraticRoots program...");
    } //run
} //class
public boolean hasRealRoots(double a, double b, double c) {
    if (b*b - 4*a*c >= 0) { return true; }
    else { return false; }
} //hasRealRoots

calcl public double calcRoot1(double a, double b, double c) {
    return (-b + Math.sqrt(b*b-4*a*c))/(2*a);
} //calcRoot1

calcl public double calcRoot2(double a, double b, double c) {
    return (-b - Math.sqrt(b*b-4*a*c))/(2*a);
} //calcRoot2

The core of the program is contained in the lines copied below. Note how simple this is to read and understand due to the use of methods, which by push the real work off to the hasRealRoots( ), calcRoot1( ) and calcRoot2( ) methods.

if (hasRealRoots(a, b, c) == true) {
    x1 = calcRoot1(a, b, c);
    x2 = calcRoot2(a, b, c);
    println("x1: " + x1 + "  x2: " + x2);
}

In this small program writing the methods might not be worth the trouble. But in a longer program that handles many quadratics (perhaps from getting data from a file or as the result of other calculations), the hasRealRoots( ), calcRoot1( ) and calcRoot2( ) methods might be used dozens of times, making the effort to write them pay off.

8.1.4 char Methods and Primitive Data Types

Methods using chars can very useful. For example, below is a method that compares two chars, using the ordering in the ASCII table, and returns the least of these two.

public char minChar(char x, char y) {
    if (x < y) return x;
    else return y;
} //minChar
And here’s a method that uses type casting to return the `char` associated with a particular integer, based on the ASCII table.

```java
public char intToChar(int x)
{
    return (char) x;
}
```

For the integer 57 this method returns the character ‘9’, for 94 it returns ‘^’ and for 120 it returns ‘x’.

### 8.1.5 More About Methods and Primitive Data Types

There are additional facts to know about these methods.

- Methods can be written that return the other primitive data types—`byte`, `short`, `long` and `float`. They work the same as the methods we've already considered so no further discussion is needed.
- A method is a block and so variables and constants declared within a method, including within the argument list, are local to that function, meaning that they exist only while that method is executing and are not accessible outside of that method.
- A method may have more than one `return` statement.
  ```java
  public int compareChars(char c1, char c2)
  {
      if (c1 < c2) return -1;
      else if (c1 == c2) return 0;
      else return 1;
  }
  ```
- A method that returns a value must return a value in every branch of execution within the method. For example the method
  ```java
  public int compareInts(int v1, int v2)
  {
      if (v1 < v2) return -1;
      else if (v1 == v2) return 0;
  }
  ```
  will not compile because one possible path of execution, occurring when `v1 > v2`, does not return a value. This error can be very difficult for the programmer to find when the method is long and complicated.
- A method may return one type only. It is not possible to write a method with one branch of execution that returns a `char` and another that returns a `boolean`.
- A method may return no more than one value.

### 8.2 Methods and Objects

Passing objects to methods and returning them from methods is just as useful so let’s look at that now.
8.2.1 Objects as Arguments

The Rule for Argument Passing tells us that objects are passed by reference. Let’s take a look at what that means.

Changing an object in a method

As a consequence of the Rule for Argument Passing changes to objects inside a method do change the arguments outside the method. Here’s an example.

```java
G Oval o1 = new G Oval(0, 0);
setupOvals(o1, 200, 200, 30, 50, Color.BLUE, true);
add(o1);
```

```java
public void setupOvals(G Oval o, int xLoc, int yLoc, int xSize, int ySize, Color cv, boolean fill)
{
    o.setLocation(xLoc, yLoc);
    o.setSize(xSize, ySize);
    o.setColor(cv);
    o.setFilled(fill);
}
```

Not changing an object in a method

But we don’t have to change an object when we pass it as an argument. If `o1` is a `G Oval` and `r1` is a `G Rect`, we might want a method that checks them for having the same color but leaves them unchanged.

```java
public boolean isSameColor(G Oval o, G Rect r)
{
    if (o.getColor() == r.getColor()) return true;
    else return false;
}
```

Can we write a version that works for any of the ACM Java graphic objects? You bet, and here it is.

```java
public boolean isSameColor(Object o, Object r)
{
    if (o.getColor() == r.getColor()) return true;
    else return false;
}
```

This works because the `G Objects` `o` and `r`, being a `G Oval` and `G Rect` respectively, each have their own `getColor()` methods. Can we apply the `isSameColor()` method to objects like `UFOs`?
that we've created? Yes, if we've included a `getColor()` method with the `UFO` class it will work. And we'll cover how to do that in the next chapter.

**Numeric methods and objects**

Objects passed to methods can be used to return numeric values. Here’s a little example.

```java
public double calcRectArea(GRect r) {
    return r.getWidth() * r.getHeight();
}
```

Not surprisingly methods with object parameters can just as easily return the other primitive numeric data types.

**Boolean methods and objects**

Objects can also be used with boolean methods. Perhaps we need to test two `GRects` for equality. We’d like to use the test like this:

```java
if (isEqual(rect1, rect2) == true) doSomething;
```

But what does it mean to say two `GRects` are equal? That depends on the application. In this case let’s define two `GRects` to be equal if and only if

- they are the same color—we need the `getColor()` method, described in the Quick Reference
- they are the same dimensions
- they have the same filled/not filled status—we need the `isFilled()` method, see the Quick Reference

which leads us to this method

```java
public boolean isEqual(GRect r1, GRect r2) {
    if ((r1.getColor() == r2.getColor())
        && (r1.getWidth() == r2.getWidth())
        && (r1.getHeight() == r2.getHeight())
        && (r1.isFilled() == r2.isFilled()))
        return true;
    else
        return false;
}
```

**Char methods and objects**

Go back to the `QuadraticRoots` program and consider these statements.

```java
answerLine = readLine("Again? y/n ");
answer = answerLine.charAt(0);
```
readLine() extracts a line of keyboard input which is then assigned to the String answerLine. Then the charAt(0) copies the first character from answerLine and assigns it to the char answer.

This wouldn't be needed if the ACM Library included a readChar() method, but it doesn't. So we'll write our own.

public char readChar(String phrase)
{
    String answer;
    answer = readLine(phrase);
    return answer.charAt(0);
} //readChar

readChar() returns a char and is therefore a char method.

In the QuadraticRoots program the two statements at the top of this section can be replaced with

answer = readChar("Again? y/n");

and the declaration of answerLine as a String can be deleted because we don’t need answerLine anymore. The result is cleaner, more easily understood code. readChar() might be a useful method to keep handy.

8.2.2 Objects as Returned Values

Not surprisingly methods can return objects just as easily as they return primitive data types.

GRect r1;
r1 = setupRects(300, 300, 80, 20, Color.BLUE, false);
add(r1);

public GRect setupRects(int xLoc, int yLoc, int xSize, int ySize, Color cv, boolean fill)
{
    GRect r = new GRect(xLoc, yLoc, xSize, ySize);
    r.setColor(cv);
    r.setFilled(fill);
    return r;
} //setupRects

You probably realize that when we return an object what we are really returning is the address in memory of that object. Thus assigning the address of r to r1, as done above, has exactly the intended effect. r1 points to a new GRect object with the desired characteristics.

©Daniel L. Schuster, 2009, all rights reserved
One last thing—\textit{r} in the \texttt{setupRects()} is a local variable. It disappears as soon as the method has executed. That's okay because \textit{r} isn't actually the object (the rectangle created inside the method). It's a pointer to the object, meaning it contains the address of the object. \textit{r} disappears but the object doesn't. It's still there and is now pointed to by \textit{r1} so we haven't lost it.

What if we did somehow lose the object? We'll talk about that in the next chapter.

\textbf{8.2.3 More About Methods and Objects}

Methods that return objects work very much like methods that return a primitive data type.

- Methods can return any object—standard Java objects such as \texttt{Strings} or \texttt{Characters}, ACM Java objects such as \texttt{GRects} or \texttt{GObjects}, and programmer defined objects like \texttt{UFOs}.
- A method is a block and so objects declared within a method are local to that method.
- A method may have more than one \texttt{return} statement.
- A method that returns an object must return an object in every branch of execution within the method.
- A method can return one object type only. It is not possible to write a method with one branch of execution that returns a \texttt{GRect} and another that returns a \texttt{GOval}. \textit{But there's a way around this—sometimes.} Since \texttt{GRects} and \texttt{GOvals} are both \texttt{GOBJECTs} it would be legitimate to have

\begin{verbatim}
public GObject myMethod(argument list)
{
    if (boolean condition)
    {
        GRect r = new GRect(arguments);
        return r;
    }
    else
    {
        GOval o = new GOval(arguments);
        return o;
    }
} //myMethod
\end{verbatim}

This method is still returning one object type, \texttt{GObject}.
- A method may return no more than one object.

\textbf{8.2.4 Example Program}

Let's develop a version of Pong, a classic arcade game. Start with a visual layout like this
Consider the following algorithm that describes basic game play, bouncing the ball around the table and allowing the player to hit it with the paddle.

```c
init method
{
    set up paddles
    set up ping pong ball
    set up table
}

mousemove method
{
    move player paddle with mouse
}

game loop
{
    make computer paddle follow the ping pong ball
    if (computer paddle hits ping pong ball)
        reverse x direction
    else if (player paddle hits ping pong ball)
        reverse x direction
    else if (ball hits the left edge)
        reverse x direction
    else if (ball hits top edge)
        reverse y direction
    else if (ball hits bottom edge)
        reverse y direction
    move the ball
    pause
}
```

Statements like *make computer paddle follow the ping pong ball* should be translated to method calls such as
followBall(compPaddle);

which will move the computer paddle up and down so that it follows the ping pong ball automatically. There are two checks for collision between objects, so we should create a boolean method for checking for collision. The collision checks become

```java
if (collides(paddle, ball) == true) do something
```

Hitting the edge of the window is handled in methods by checking coordinates. Finally, we'll skip scoring for now and just run our game in an infinite loop.

Here's a starter solution that implements this basic play. It's a straightforward translation of the algorithm.

```java
BasicPongWithMethods
//BasicPongWithMethods.java
import acm.program.*;
import acm.graphics.*;
import acm.util.*;
import java.awt.*
import java.awt.event.*;

public class BasicPongWithMethods extends GraphicsProgram {
    public static final int APPLICATION_WIDTH = 400;
    public static final int APPLICATION_HEIGHT = 300;

    final int APPW = APPLICATION_WIDTH;
    final int APPH = APPLICATION_HEIGHT;
    final int WAIT1 = 5;
    final int BSIZE = 20; //ball diameter
    final int PSIZE = 50; //paddle length

    GOval ball;
    GRect compPaddle, playerPaddle;

    int ballXPos, ballYPos;

    public void init() {
        pingPongBallSetup();
        paddleSetup();
        setBackground(Color.GREEN);
        addMouseListeners();
    }
```

what's up with these two statements?

these are global constants and variables so they can be used anywhere in the program.
public void mouseMoved(MouseEvent e)
{
    double y = e.getY();
    playerPaddle.setLocation(10, y-BSIZE/2);
} //mouseMoved

public void run()
{
    int jump = 2, xMove = -jump, yMove = jump;

    while(true) //game loop
    {
        followBall(compPaddle);

        if (collide(ball, compPaddle) == true)
        {
            xMove = -xMove;
        }
        else if (collide(ball, playerPaddle) == true)
        {
            xMove = -xMove;
        }
        else if (hitsLeftEdge(ball) == true) xMove = -xMove;
        else if (hitsTopEdge(ball) == true) yMove = -yMove;
        else if (hitsBottomEdge(ball) == true) yMove = -yMove;

        ball.move(xMove, yMove);
        pause(WAIT1);
    } //game loop
} //run

public void pingPongBallSetup()
{
    ball = new GOval(BSIZE, BSIZE);
    ball.setFilled(true);
    ball.setColor(Color.WHITE);
    RandomGenerator rg = new RandomGenerator();
    ballXPos = rg.nextInt(50, APPW-50);
    ballYPos = rg.nextInt(0, APPH);
    add(ball, ballXPos, ballYPos);
} //pingPongBallSetup

public void paddleSetup()
{
}
compPaddle = new GRect(5, PSIZE);
compPaddle.setColor(Color.BLUE);
compPaddle.setFilled(true);
add(compPaddle, APPW-10, ballYPos-PSIZE/2);

playerPaddle = new GRect(5, PSIZE);
playerPaddle.setColor(Color.RED);
playerPaddle.setFilled(true);
add(playerPaddle, 10, APPH/2-PSIZE/2);

public void followBall(GRect compPaddle)
{
    compPaddle.setLocation(APPW-10, ball.getY()-10);
}

public boolean collide(GOval b, GRect paddle)
{
    GRectangle bBox = b.getBounds();
    GRectangle pBox = paddle.getBounds();
    if (bBox.intersects(pBox) == true) return true;
    else return false;
}

public boolean hitsLeftEdge(GOval ball)
{
    if (ball.getX() < 0) return true;
    else return false;
}

public boolean hitsTopEdge(GOval ball)
{
    if (ball.getY() < 0) return true;
    else return false;
}

public boolean hitsBottomEdge(GOval ball)
{
    if (ball.getY() + BSIZE > APPH) return true;
    else return false;
}

A more advanced version of the game would need
- instructions
- a better looking table and paddles
- scoring
Problem Set

1) **testCC** - the LUHN Formula for Validation of Primary Account Number - the following steps are required to validate a credit card number:
   a) Double the value of alternate digits of the primary account number beginning with the second digit from the right (the first right-hand digit is the check digit.)
   b) Add the individual digits comprising the products obtained in Step 1 to each of the unaffected digits in the original number.
   c) If the total obtained in Step 2 is a number ending in zero (30, 40, 50, etc.) the account number is valid.

For example, to validate the primary account number 49927398716:
   a) 
   
   \[
   \begin{array}{ccccccccc}
   4 & 9 & 9 & 2 & 7 & 3 & 9 & 8 & 7 & 1 & 6 \\
   \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \\
   \end{array}
   \]

   \[
   \begin{array}{cccc}
   4 & 18 & 9 & 4 & 7 & 6 & 9 & 16 & 7 & 2 & 6
   \end{array}
   \]

   b) \(4 + (1+8) + 9 + (4) + 7 + (6) + 9 + (1+6) + 7 + (2) + 6\)
   c) Sum = 70 : Card number is validated.

Write a program including a method **public boolean isValidCC(String ccNumber)** that tests **ccNumber** to see if it represents a valid credit card number. You may want to write several other helper methods. Test it with the above number and with your own credit cards, as well as 5105105105105100 and 4111111111111111, both of which should be valid. Now try it with the same numbers, but change any one of the digits.

2) **Pong2WithMethods** Redo the Pong2 assignment from Chapter 6, this time using methods as they were used in the BasicPongWithMethods above.

2) **HelicopterInACaveWithMethods** Rewrite the HelicopterInACave game from chapter 6, using methods where reasonably possible, including checking for collision of the helicopter and the cave objects.

3) **BugBotWithMethods** Rewrite the Bugbot from Chapter 6, using methods where reasonable.

4) **UFOEscapeWithMethods** Rewrite the UFOEscape from Chapter 6, using methods.
5) **YourChoiceWithMethods** Rewrite your original YourChoice program using methods where appropriate. Your instructor's approval is required before you begin.

6) **PigLatin** Pig Latin is a silly 'language' based on English but translated using the following scheme.
   a. words that begin with a consonant – the consonant and all characters up to (but not including) the first vowel are removed and added as a suffix along with the string "ay" great  eat-gray
   b. words that start with a vowel – have the string "ay" added as a suffix – and  and-ay

Write a console program, using methods where reasonable, that inputs a string and displays its Pig Latin equivalent.

| Enter sentence: You speak great Pig Latin. |
| Enter Sentence: This is a hard project. |
| Translation: is-thay is-ay a-ay ard-hay oject-pray. |