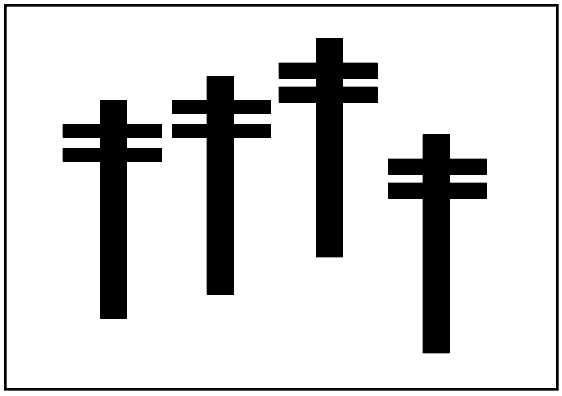
Chapter 2 Repetition

2.1 Repetition — Methods with parameters

Suppose we want to produce the following image.



We might start out with the following program that draws only the left most object.

```
import java.awt.*;

public class Poles extends java.applet.Applet
    {
    public void paint(Graphics g)
        {
        paintPhonePoleOne(g);
        }

    private void paintPhonePoleOne(Graphics g)
        {
        g.setColor(Color.black);
        g.fillRect(40, 40, 10, 120);
        g.fillRect(20, 50, 50, 5);
        g.fillRect(20, 65, 50, 5);
        }
    }
```

We might add a second method, *paintPhonePoleTwo*, whose graphics commands are almost the same, except that the left side of each rectangle is moved over 80 spaces and the top of each rectangle is moved up 10 spaces. This results in the following method:

```
    private void paintPhonePoleTwo(Graphics g)
        {
        g.setColor(Color.black);
        g.fillRect(120, 30, 10, 120);
        g.fillRect(100, 40, 50, 5);
        g.fillRect(100, 55, 50, 5);
        }
```

If we have made no errors in changing the numbers in the first two columns, this method will paint the second phone pole.

However, changing the numbers is time consuming and error prone. The computer is happy to do the arithmetic and is unlikely to make errors. It's probably better to write this method by copying the original method and have the computer add 80 and subtract 10 appropriately. This gives an alternate version of *paintPhonePoleTwo*.

```
private void paintPhonePoleTwo(Graphics g)
    {
    g.setColor(Color.black);
    g.fillRect(40 + 80, 40 - 10, 10, 120);
    g.fillRect(20 + 80, 50 - 10, 50, 5);
    g.fillRect(20 + 80, 65 - 10, 50, 5);
    }
```

Choosing different numbers to add and subtract will give usable methods to paint the remaining poles.

It would be much better if we could make a single, generic method to draw a phone pole. In fact, it would be very convenient if the *paint* method could be written as

```
public void paint(Graphics g)
    {
    paintPhonePole(g, 20, 40);
    paintPhonePole(g, 100, 30);
    paintPhonePole(g, 180, 20);
    paintPhonePole(g, 260, 50);
    }
```

where the numbers on each line represent the positions of the left side and the top of each phone pole.

A method can be written to accept numbers if we write it as shown here.

```
private void paintPhonePole(Graphics g, int left, int top)
    {
    g.setColor(Color.black);
    g.fillRect(left + 20, top, 10, 120);
    g.fillRect(left, top + 10, 50, 5);
    g.fillRect(left, top + 25, 50, 5);
    }
```

The word **int** is Java's abbreviation for integer. Used in this way, the words *left* and *top* are standins for the values supplied in the *paint* method. These standins are called *formal parameters*. The values the *formal parameters* are given are the numbers in the command that starts the method. The numbers given in the command are called *actual parameters*.

We can now write a compact program to create the image with four phone poles.

```
import java.awt.*;

public class Poles extends java.applet.Applet
    {
    public void paint(Graphics g)
        {
        paintPhonePole(g, 20, 40);
        paintPhonePole(g, 100, 30);
        paintPhonePole(g, 180, 20);
        paintPhonePole(g, 260, 50);
        }
    private void paintPhonePole(Graphics g, int left, int top)
        {
        g.setColor(Color.black);
        g.fillRect(left + 20, top, 10, 120);
        g.fillRect(left, top + 10, 50, 5);
        g.fillRect(left, top + 25, 50, 5);
        }
    }
```

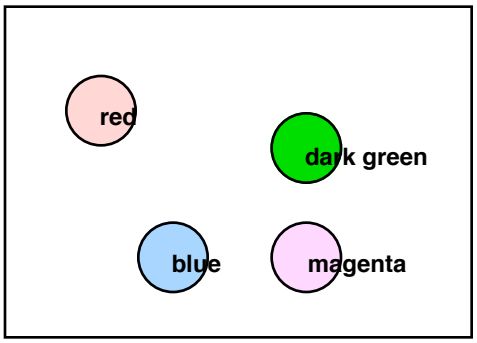
Parameter types

The items in parentheses that are provided to a method (as the initial values for *formal parameters*) are called *actual parameters*.

Several immediately useful kinds of parameters are available. These include *int*, *Color*, and *String*.

- *int* is used when the parameter will be an integer.
- *Color* is used when the parameter will be a color; any item that can be used in `g.setColor(…)`.
- *String* is used when the parameter will be a word or words surrounded by double quotes; the word "Hi" in the first example in chapter 1 is an example.

As an example, the following program places several different colored circles on the screen along with the names of the colors.

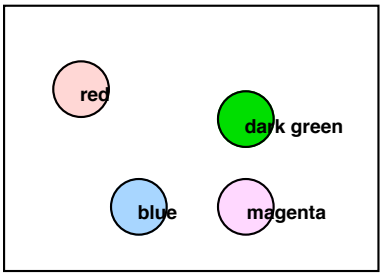



```
import java.awt.*;

public class Circles extends java.applet.Applet
   {
   public void paint(Graphics g)
      {
      paintCircle(g, 20, 20, Color.red, "red");
      paintCircle(g, 40, 60, Color.blue, "blue");
      paintCircle(g, 100, 60, Color.magenta, "magenta");
      paintCircle(g, 100, 30, new Color(0, 150, 0), "dark green");
      }

   private void paintCircle(Graphics g, int x, int y, Color theColor, String name)
      {
      g.setColor(theColor);
      g.fillOval(x - 10, y - 10, 20, 20);
      g.setColor(Color.black);
      g.drawString(name, x, y + 4);
      }
   }
```

The Signature of a Method

The information on the first line of a method is collectively called the *signature* of the method.
The signature of *paintPhonePoleOne* indicates that a *Graphics* object is required.
The signature of *paintPhonePole* indicates that a *Graphics* object and two integers are required.
The signature of *paintCircle* indicates that a *Graphics* object, two integers, a color and a string are required.

The Console Window




```
a red circle at x = 20  y = 20
a blue circle at x = 40  y = 60
a maganta circle at x = 100  y = 60
a dark green circle at x = 100  y = 30
```

The Console Window will display the value of variables if the parameter to *System.out.println* is a 'sum' consisting alternately of strings and variables. For example, to have the values *x* and *y* printed out for each circle, put the following line into the *paintCircle* method:

```
System.out.println("a " + name + " circle at x = " + x + "  y = " + y);
```

Try this out. Obtaining a running commentary on what your program is doing is an important skill.

Concatenation: creating long strings from shorter ones

When Java is asked to 'add' a string to something else, Java puts both items together to create a longer string. Putting two strings together to create a longer string is called *concatenation*. When the '+' symbol is used to put strings together, it is called the *concatenation operator*. '+' symbols are used in this way to produce the console window output above. Generally, if the are numbers on both sides of '+', addition is done, if one or both of the items adjacent to '+' is a String, concatenation is done.

Variables – types, declaration & initialization

A name that is used to represent a data value is called a *variable*. Formal parameters are variables. We can also create names for data values without using them as a formal parameters. This is done when we want a reference to something (a number, a color, etc.) whose value can be changed during the execution of a program. We will do this in the next section of the text.

There are eight *scalar variable types* and a virtually unlimited number of types which are defined by *classes*. Each scalar variable holds a single data value. The scalar types most commonly used are *int* (holds an integer), *double* (holds an integer or a fractional value), *char* (holds a keyboard character), and *boolean* (holds true or false). Types which are defined by classes will be discussed in detail as the need arises.

When you first use a name that is to be that of a variable you must state its type. This is called *declaring* the variable. You should then assign the variable a starting value. This is called *initializing* the variable. Formal parameters are initialized using the values of actual parameters. The following examples give two equivalent forms for initializing variables that are not formal parameters.

```
int aNumber;              // aNumber, a variable of type integer, is declared on this line.
aNumber = 3;              // The starting value assigned to aNumber is 3. aNumber is
                          //      initialized on the second line.

int aNumber = 3;          // both declaration and initialization are done in one line.
```

Exercise — 2.1

Write a program with several repeated images. Have two (or more) different kinds of images. Have the *paint* method only use the names of other methods. Place the actual instructions for creating the images in the other methods.

2.2 Many Repetitions — Loops

If we have a need to repeat a process many times, there is usually a simple pattern to the parameters. In the example below, we draw nine lines, each ten dots below the previous line.

```
import java.awt.*;

public class Lines extends java.applet.Applet
   {
   public void paint( Graphics g )
      {
      drawLines(g);
      }
   private void drawLines( Graphics g )
      {
      g.setColor(Color.black);
      g.drawLine(10,  10, 100,  10);
      g.drawLine(10,  20, 100,  20);
      g.drawLine(10,  30, 100,  30);
      g.drawLine(10,  40, 100,  40);
      g.drawLine(10,  50, 100,  50);
      g.drawLine(10,  60, 100,  60);
      g.drawLine(10,  70, 100,  70);
      g.drawLine(10,  80, 100,  80);
      g.drawLine(10,  90, 100,  90);
      g.drawLine(10, 100, 100, 100);
      }
   }
```

Since we have a very systematic change in the coordinates for each line, we should be able to tell the computer what the pattern is and thus avoid typing so many similar commands. One of the most common methods for specifying numerical patterns to a computer is the structure described below.

The *while* Control Structure

There are several *control structures* included in the Java language, *while* and *if* (discussed in chapter 3) are the most basic and most important.

The *while* structure is used to repeatedly execute one or more commands. This is called looping and the structure itself is often called a loop.

The form of the *while* structure is the word "while" followed by an expression which is either true or false. This is followed by the commands to be executed.

The two forms of the while structure are as follows:

```
while (true or false expression)
   command;
```

or

```
while (true or false expression)
   {
   command;
   command;   // any number of commands – a group of commands with braces is called a block.
   command;
   }
```

To use the *while* structure to execute the commands in the *drawLines* example we first write a generic version of the command to be repeated.

```
g.drawLine(10, row, 100, row);
```

Every one of the commands in the example is of this form with the variable *row* replaced by one of the numbers 10, 20 … 100.

The complete structure (along with a required initialization step that precedes the *while* structure itself) looks like this:

```
int row;                                  Declare that the variable row is an integer.
row = 10;                                 Set row equal to 10.
while (row <= 100)                        Repeat the commands as long as row is less than or
    {                                       equal to 100.
    g.drawLine(10, row, 100, row);        Draw a line.
    row = row + 10;                       Increase the value of row by 10.
    }
```

And the whole program becomes:

```
import java.awt.*;

public class Lines extends java.applet.Applet
   {
   public void paint( Graphics g )
      {
      drawLines(g);
      }

   private void drawLines( Graphics g )
      {
      g.setColor(Color.black);
      int row = 10;
      while (row < 100)
         {
         g.drawLine(10, row, 100, row);
         row = row + 10;
         }
      }
   }
```

In the program the two lines *int row;* and *row = 10;* are written in one line. The one line form is generally the best form to use; two lines were used above to allow room for comments. The two forms are equivalent.

Generally, the use of a *while* structure involves 4 distinct program parts:

- initialization
- testing
- the body of the loop
- incrementation

The following are the parts for the above example:

```
int row;                                initialization
row = 10;
while (row < 100)                       testing
   {
   g.drawLine(10, row, 100, row);       body
   row = row + 10;                      incrementation
   }
```

It is easy to create a *while* structure which will not terminate. It is amazingly common to simply forget the incrementation step. The example below (which uses '!=' meaning 'not equal') is more subtle.

```
int row = 15;
while (row != 50)
   {
   g.drawLine(10, row, 100, row);
   row = row + 10;
   }
```

The row value starts at 15, and then takes on the values: 25, 35, 45, 55, ... skipping right over the desired value to determine termination of the block. It is important to pay special attention to the expression to be tested for truth and the mechanism for modifying the truth of the expression.

Important notes on 'infinite loops'

If the expression never becomes false, the computer will 'get stuck' performing the commands over and over, and the program will appear to 'freeze'. This is a very common error and is called an infinite loop (various Anglo-Saxon monosyllables are also commonly used).

NO SEMI-COLON belongs at the end of the line containing *while*! This is a common error and will result in an 'infinite loop'. This is an insidious 'anti-feature' of Java (and numerous programming languages that have preceded it). Every programmer has lost hours (if not days) cursing a non-functioning program whose major error is the existence of a semi-colon at the end of a line starting with *while*.

Incrementing Variables

There are several ways to write the command to increase the value of a variable. These include:

1. row = row + 10;
2. row += 10;
3. row++;

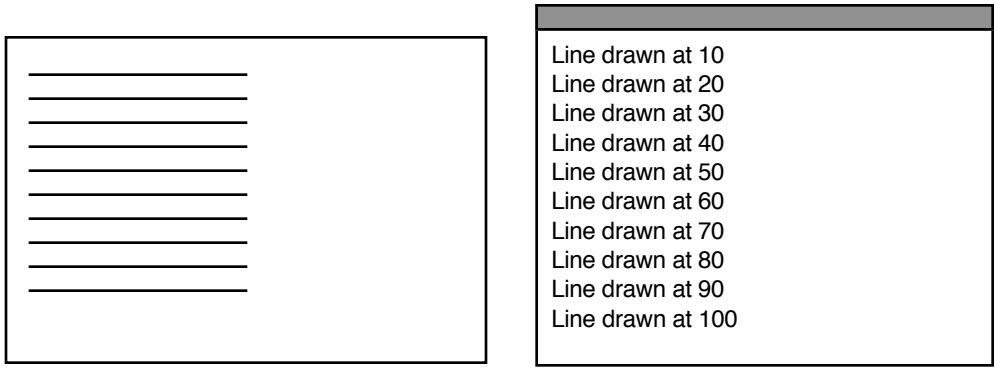
Command '1' makes a copy of row, adds 10 to the copy, then reassigns the new value to row. The effect is simply to add 10 to row.

Command '2' is a shortened form of the command above; it does the same thing as Command '1' .

Command '3' increments 'row' by one. One is the most common amount to add, and this shorthand form is very widely used.

Each of the above can be used to subtract a number; just change the '+' symbols to '–'.

The Console Window



As soon as programs contain *while* structures, the console window becomes an important tool for correcting errors. To have a loop report its progress, one places a printing command inside of the loop. For example, to have the value of *row* printed out in the example in this section, put the following line inside the *while* loop in the *drawLines* method:

```
System.out.println("Line drawn at " + row);
```

Scope and lifetime of a variable

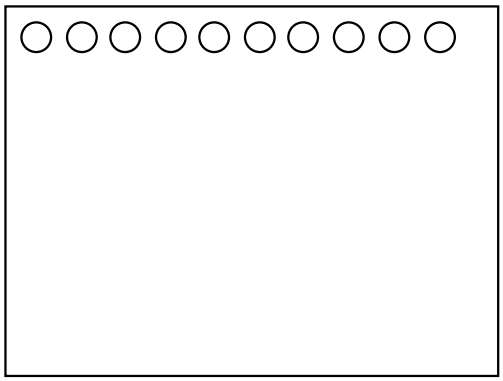
It is important to know where and how long a variable "lives"; what parts of the program can see it, use it, or change it; and how long it will exist. These properties of a variable are usually referred to by the terms *scope* of a variable and *lifetime* or *persistence* of a variable.

- Variables declared in a method are created when the method starts and are destroyed when the method is finished. The *lifetime* of a variable declared in a method is the period of time during which the method is executing.
- A variable cannot be seen outside of the method in which it is declared. If a variable named *row* is declared in each of several methods, there is a different variable named *row* in each method. The *scope* of a variable declared in a method is the method in which it is declared.
- Variables of the types *int*, *Color*, and *String* which we have been using cannot be changed outside of the method in which they are declared – even if they are sent as parameters to another method. Some types of variables can be changed outside of the method in which they are declared. This type of variable is a reference to an 'object with mutators'. The only 'object with mutators' we have encountered up to this point is the *Graphics* object that appears in the *paint* methods. We will not encounter other objects with mutators until chapter 3, section 9.

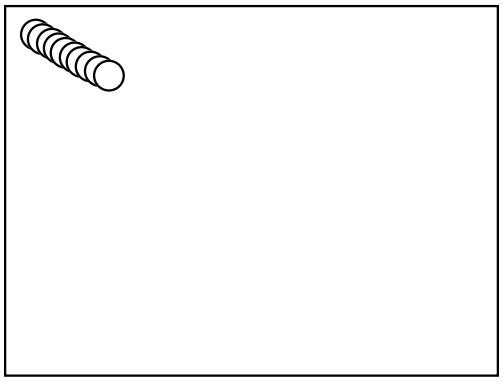
Graphical Examples

The methods below produce the illustrated images. They need to be included in a framework of an actual complete program. To make them work, replace the *drawLines* method in the previous example with one of these methods and call the new method within the *paint* method.

```
private void rowOfCircles(Graphics g)
    {
    int howMany = 10;
    int painted = 0;
    int column = 10;
    while (painted < howMany)
        {
        g.drawOval(column, 20, 10, 10);
        column = column + 15;
        painted++;
        }
    }
```



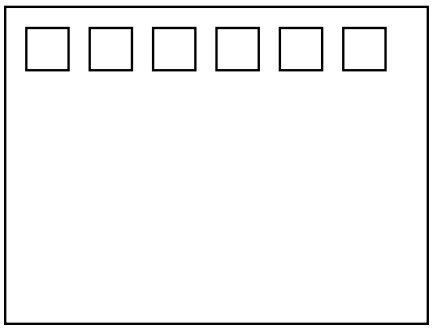
```
private void pileOfCircles(Graphics g)
    {
    int howMany = 10;
    int painted = 0;
    int column = 10;
    int row = 10;
    while (painted < howMany)
        {
        g.setColor(Color.white);
        g.fillOval(column, row, 10, 10);
        g.setColor(Color.black);
        g.drawOval(column, row, 10, 10);
        column = column + 3;
        row = row + 2;
        painted++;
        }
    }
```



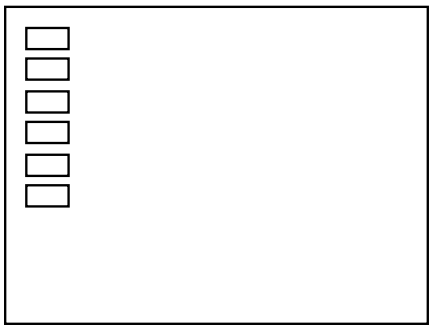
Exercises — 2.2

For each figure, write a method containing a loop that will create a similar figure:

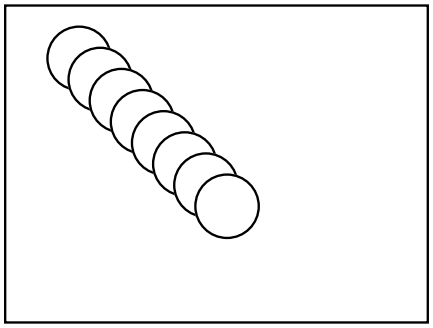
1.



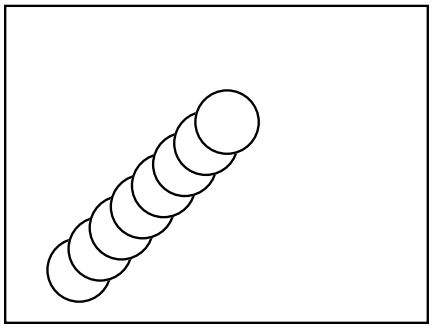
2.



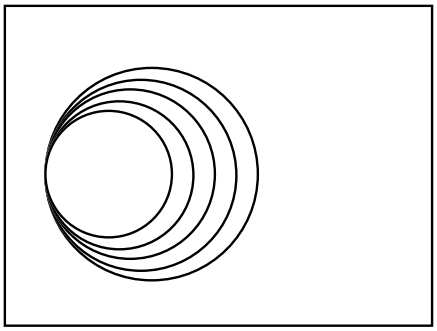
3.



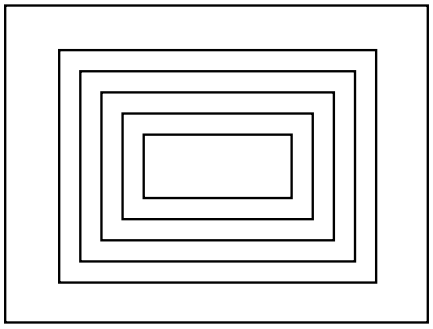
4.



5.



6.



Possible answer for (1.)

```
private void boxRow(Graphics g)
    {
    int column = 10;
    int count = 0;
    while (count < 6)
        {
        g.drawRect(column, 10, 20, 20);
        System.out.println("Rect drawn at " + column);  // reports drawn location
        column += 25;
        count++;
        }
    }
```

2.3 Expressions

True and false expressions

There is a consistent difference in meaning between '=' and '=='.

The single = is used when the intent is to force two things to be equal (the item on the left is changed when necessary).

The double == is used when the intent is to inquire as to the equality of two things. That is, the statement a == b is a statement which has truth value associated with it. Either a and b are the "same" and the statement a == b is true, or a and b are not the same and the statement a == b is false. The truth or falseness of a statement such as this is generally used to determine what the program will do next.

The most common true or false expressions are of the form:

`x == 4` (x is equal to 4?)	NOTE: $x = 4$ makes x equal to 4, it is NOT a true or false expression
`x < 4`	x is less than 4
`x <= 4`	x is less than or equal to 4
`x > 4`	x is greater than 4
`x >= 4`	x is greater than or equal to 4
`x != 4`	x is not equal to 4

Arithmetic expressions

The most common arithmetic expressions are of the form:

`x + 4`	x plus 4
`x - 4`	x minus 4
`x * 4`	x times 4
`x / 4`	x divided by 4

If x is an integer (as in the examples in this chapter), x divided by 4 is a integer. The remainder (if any) is discarded so that this will be so.

`x % 4`	x modulo 4 (this is the remainder upon dividing by 4)

Since we are dividing by 4, the result of this expression will be 0, 1, 2 or 3 (0, –1, –2, or –3 if x is negative).

Constants

If a value is used repeatedly, or the program will be more readable with a name for a value, a *constant* is created. Constants are given names which are written entirely in capital letters, to aid in recognition of their status. Constants are declared and initialized in the same statement.

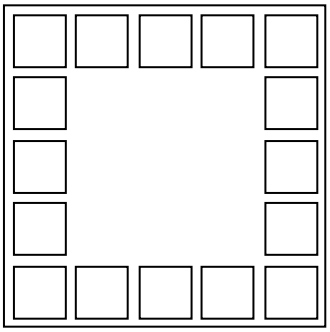
The format for assigning a value to a constant is as follows:

```
private static final int WIDTH = 20;
private static final Color DARK_GREEN = new Color(0, 120, 0);
```

Constants should be declared before the first method begins. Constants cannot be declared inside of methods.

Constants with multi-word names are written with an underscore '_' between each word.

2.4 Using *while* with parameters



If you wanted to make a pattern of boxes such as that show here, you could use brute force and draw each box one at a time.

Brute force is a poor technique, especially if the number of items is large. We will create this pattern using the while control structure. If we look at the figure as containing a row of boxes at the top, a row at the bottom, a column to the left, and a column to the right, we can see a way to use four while loops to draw the four lines of boxes. Putting each loop into a separate method, we get the following program:

```
import java.awt.*;

public class BoxBorder extends java.applet.Applet
    {
    private static final int BOXES = 5;
    private static final int WIDTH = 20;
    private static final int HEIGHT = 20;

    public void paint(Graphics g)
        {
        paintTopRow(g);
        paintBottomRow(g);
        paintLeftColumn(g);
        paintRightColumn(g);
        }

    private void paintTopRow(Graphics g)
        {
        int b = 0;
        int left = 5;
        int top = 5;
        while (b < BOXES)
            {
            g.drawRect(left, top, WIDTH, HEIGHT);
            left += WIDTH + 5;
            b++;
            }
        }

    private void paintBottomRow(Graphics g)
        {
        int b = 0;
        int left = 5;
        int top = 105;
        while (b < BOXES)
            {
            g.drawRect(left, top, WIDTH, HEIGHT);
            left += WIDTH + 5;
            b++;
            }
        }
```

Program continues on the next page.

```
    private void paintLeftColumn(Graphics g)
        {
        int b = 2;
        int left = 5;
        int top = 30;
        while (b < BOXES)
            {
            g.drawRect(left, top, WIDTH, HEIGHT);
            top += HEIGHT + 5;
            b++;
            }
        }
    private void paintRightColumn(Graphics g)
        {
        int b = 2;
        int left = 105;
        int top = 30;
        while (b < BOXES)
            {
            g.drawRect(left, top, WIDTH, HEIGHT);
            top += HEIGHT + 5;
            b++;
            }
        }
    }
```

Since the *private* methods are so similar, it seems natural to create fewer methods, each with an additional parameter . Since the only difference between the rows is the distance of the boxes from the top of the window, a parameter can be used for the top. Similarly with the columns, the left value is the only difference in the two methods. When these modifications are made, the resulting program could look as follows:

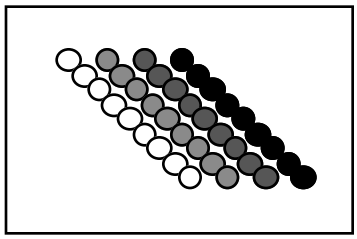
```
import java.awt.*;

public class BoxBorder extends java.applet.Applet
    {
    private static final int BOXES = 5;
    private static final int WIDTH = 20;
    private static final int HEIGHT = 20;

    public void paint(Graphics g)
        {
        paintRow(g, 5);
        paintRow(g, 105);
        paintColumn(g, 5);
        paintColumn(g, 105);
        }

    private void paintRow(Graphics g,int top)
        {
        int b = 0;
        int left = 5;
        while (b < BOXES)
            {
            g.drawRect(left, top, WIDTH, HEIGHT);
            left += 25;
            b++;
            }
        }
```

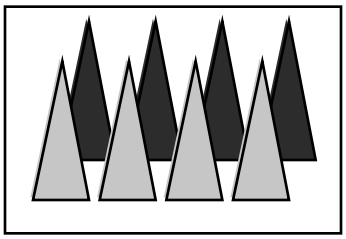
Program continues on the next page.

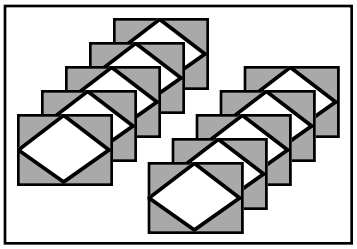
```
    private void paintColumn(Graphics g, int left)
        {
        int b = 2;
        int top = 30;
        while (b < BOXES)
            {
            g.drawRect(left, top, WIDTH, HEIGHT);
            top += 25;
            b++;
            }
        }
}
```

2.4 Exercises using parameters and *while*

Create images using methods with *while* loops and parameters. The following are intended to give you ideas.





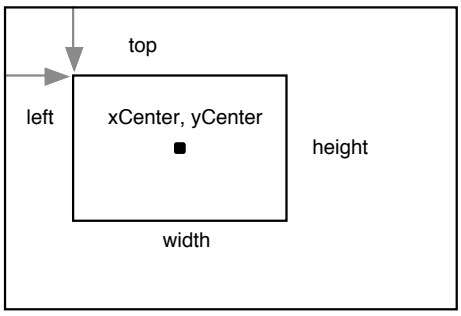


2.5 Graphics organization

Centering Graphics

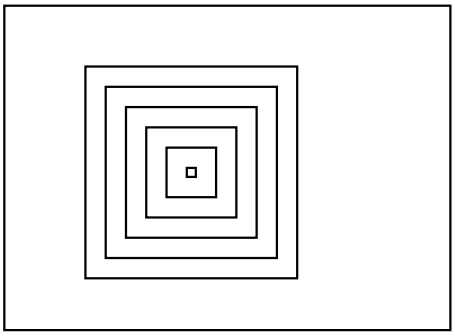
The standard command for drawing rectangles is very convenient when you know the position of the top-left corner. In many situations, it is more convenient to keep track of the center of the rectangle. The method below uses the standard command for drawing rectangles to draw a rectangle given the center point. It does this by computing the corner point from the center point, the width, and the height.

The top-left corner of the rectangle (to which the arrows are pointing), is one half of the width left of the center and one half of the height above the center. This suggests the following method:




```
private void centeredRect(Graphics g, int x, int y,
                                int width, int height)
    {
    int left = x - width/2;
    int top = y - height/2;
    g.drawRect(left, top, width, height);
    }
```

The *centeredRect* method makes the drawing of the concentric squares at the below exceptionally straightforward.



```
private void concentricSquares(Graphics g,
                           int x,int y, int howMany)
    {
    int count = 0;
    int width = 10;
    while (count < howMany)
        {
        centeredRect(g, x, y, width, width);
        count++;
        width += 10;
        }
    }
```

To obtain the concentric squares image at the right, put the following line into a *paint* method:

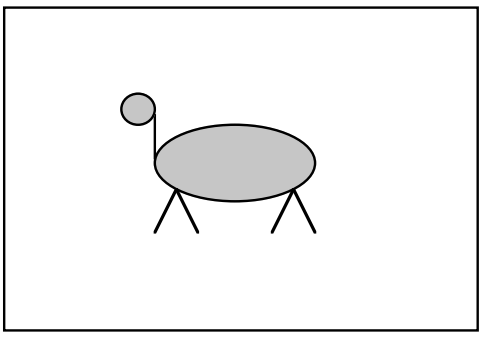
```
concentricSquares(g, 100, 100, 6);
```

Scalable Graphics

This section discusses a method for creating a method that can paint an image more complex than a single rectangle or oval at different scales without the image becoming distorted.

First we create an image that we would like to scale:

```
private void paintBeast(Graphics g)
    {
    g.setColor(Color.darkGray);
    g.fillOval(150, 150, 10, 10);
    g.fillOval(160, 160, 70, 20);
    g.drawLine(160, 155, 160, 170);
    g.drawLine(170, 177, 160, 188);
    g.drawLine(170, 177, 180, 188);
    g.drawLine(220, 177, 210, 188);
    g.drawLine(220, 177, 230, 188);
    }
```



Next we choose a convenient 'center point' from which all of the other points will be measured. The oval representing the body is created by *g.fillOval(160, 160, 70, 20)*, so the center is at (160 + 35, 160 + 10).

```
private void paintBeast(Graphics g, int x, int y)
    {
    g.setColor(Color.darkGray);
    g.fillOval(x-45, y-20, 10, 10);
    g.fillOval(x-35, y-10, 70, 20);
    g.drawLine(x-35, y-15, x-35, y);
    g.drawLine(x-25, y+7, x-35, y+18);
    g.drawLine(x-25, y+7, x-15, y+18);
    g.drawLine(x+25, y+7, x+15, y+18);
    g.drawLine(x+25, y+7, x+35, y+18);
    }
```

The center of the body is at (195, 170). We rewrite the method with each *x* and *y* coordinate measured from this point.

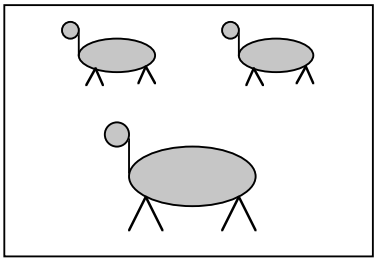
The numbers that remain unchanged are widths and heights, not *x* or *y* coordinates.

Finally we rewrite all of the remaining numbers as fractions of the size of some part of the image. The largest number remaining is 70 (the width of the body) and it is convenient to use this as the denominator for the fractions. The fractions can be reduced to lowest terms, but this is unnecessary.

```
private void paintBeast(Graphics g, int x, int y, int size)
      {
      g.setColor(Color.darkGray);
      g.fillOval(x-size*45/70, y-size*20/70, size*10/70, size*10/70);
      g.fillOval(x-size*35/70, y-size*10/70, size*70/70, size*20/70);
      g.drawLine(x-size*35/70, y-size*15/70, x-size*35/70, y);
      g.drawLine(x-size*25/70, y+size*7/70, x-size*35/70, y+size*18/70);
      g.drawLine(x-size*25/70, y+size*7/70, x-size*15/70, y+size*18/70);
      g.drawLine(x+size*25/70, y+size*7/70, x+size*15/70, y+size*18/70);
      g.drawLine(x+size*25/70, y+size*7/70, x+size*35/70, y+size*18/70);
      }
```

The method below produces three 'beasts'. A size of 70 is the full sized image, 35 gives a half-sized image.

```
private void paintBeast(Graphics g)
    {
    paintBeast(g, 120, 60, 35);
    paintBeast(g, 320, 60, 35);
    paintBeast(g, 200, 160, 70);
    }
```



2.6 Additional loop control structures

The *for* structure

The *for* structure is a shorthand way of writing a *while* structure. The two structures are almost exactly equivalent. There is one technical difference which is discussed after the first three examples.

```
for (initialization; true or false expression; incrementation)
   command;
```

or

```
for (initialization; true or false expression; incrementation)
   {
   command;
   command;                  // any number of commands
   command;
   }
```

The following structures are almost exactly equivalent:

```
1. int row = 10;
   while (row <= 100)
      {
      g.drawLine(10, row, 100, row);
      row = row + 10;
      }

2. int row;
   for (row = 10; row <= 100; row = row + 10)
      g.drawLine(10, row, 100, row);

3. for (int row = 10; row <= 100; row = row + 10)
      g.drawLine(10, row, 100, row);
```

The above examples draw the same lines on the screen. Example #3 differs from the others in that the variable *row* is not available for use in commands that might follow the *for* structure. The variable that controls a loop generally should not be used outside of the loop, thus example #3 is usually appropriate. It is certainly the most commonly used.

The use of *while* or *for* in a given program is mostly a matter of the programmer's taste. In a loop in which a variable is incremented (or decremented) and the loop is stopped when this variable reaches a certain value, the *for* structure is almost always used.

The initialization part of a *for* structure can be left blank as can the incrementation part. Thus the following example is also equivalent (although hardly reasonable).

```
4. int row = 10;
   for (; row <= 100;)
      {
      g.drawLine(10, row, 100, row);
      row = row + 10;
      }
```

The *do…while* structure

The *do…while* structure works like the *while* structure except that commands are done the first time without checking the true or false expression. After the first time, *do…while* and *while* act in the same way. The *do…while* structure is rather rarely used. When it is appropriate though, it is very handy to have.

```
do
   command;
while (true or false expression);

       or
do
   {
   command;
   command;   // any number of commands
   command;
   }
while (true or false expression);
```