Repetition, Looping

Last time we looked at how to use if-then statements to control the flow of a program. In this section we will look at different ways to repeat blocks of statements. Such repetitions are called loops and are a powerful way to perform some task over and over again that would typically be too much work to do by hand. There are several ways to construct loops. We will examine the while and for loop constructs here.

While Loop

The while loop allows you to direct the computer to execute the statement in the body of the loop as long as the expression within the parentheses evaluates to true. The format for the while loop is:

```
while (boolean_expression)
{
     statement1;
     …
     statement N;
}
```

As long as the Boolean expression evaluates to true, statements 1 through N will continue to be executed. Generally one of these statements will eventually make the Boolean expression become false, and the loop will exit. Here is an example that prints the numbers from 1 to 10:

```
int x=1;

while (x<=10)
{
     System.out.println(x);
     x++;            // Same as x=x+1
}
```

If we wanted to print out 1,000,000 numbers we could easily do so by changing the loop! Without the while loop, we would need 1,000,000 different print statements, certainly an unpleasant task for a programmer.

Here is a better version of our coin flip program that uses a while loop:

```
import java.util.Random;

public class CoinFlipDemo
{
	public static void main(String[] args)
	{
		Random randomGenerator = new Random();
		int counter = 1;

		while (counter <= 5)		// 5 flips
		{
			System.out.print("Flip number " + counter + ": ");
			int coinFlip = randomGenerator.nextInt(2);
			if (coinFlip == 1)
				System.out.println("Heads");
			else
				System.out.println("Tails");
			counter++;
		}
	}
}
```

There are two types of while loops that we can construct. The first is a *count-based* loop, like the one we just used above. The loop continues, incrementing a counter each time, until the counter reaches some maximum number. The second is an event-based loop, where the loop continues indefinitely until some event happens that makes the loop stop. Here is an example of an event-based loop:

```
	Scanner keyboard = new Scanner(System.in);
	String s;
	int sum=0, x=0;
	while (x!=-9999) {
		System.out.println("Enter an integer:");
		x = keyboard.nextInt( );
		if (x != -9999)
		{
			sum = sum + x;
		}
	}
	System.out.println("The sum of the numbers you entered is "
+ sum);
```

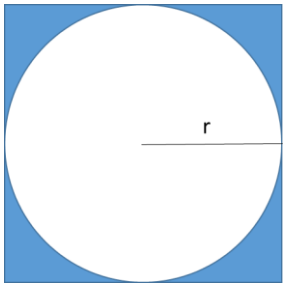
This loop will input a number and add it to sum as long as the number entered is not –9999. Once –9999 is entered, the loop will exit and the sum will be printed. This is an event-based loop because the loop does not terminate until some event happens – in this case, the special value of –9999 is entered. This value is called a *sentinel* because it signals the end of input. Note that it becomes possible to enter the sentinel value as data, so we have to make sure we check for this if we don't want it to be added to the sum.

What is wrong with the following code? Hint: It results in what is called an infinite loop.

```
int x=1, y=1;

while (x<=10)
{
      System.out.println(y);
      y++;
}
```

Coding Example – Estimating the value of Pi!

What is Pi? Where does it come from? We can simulate a physics-style experiment to estimate the value of pi!
Imagine a circle or radius r inscribed inside a rectangle.



From our knowledge of geometry we know that the area of the circle is Πr2. The area of the square is 2r*2r or 4r2. If we take the ratio of the areas we get:

$$\frac{\pi r^2}{4r^2}$$

The r^2's cancel out leaving us with the ratio equaling

$$\frac{\pi}{4}$$

This means if we can figure out the ratio of the areas, and multiply it by 4, then we would get the value of pi!

How can we get the ratio of the areas? One way is to imagine the circle and square is on a picture hung up on the wall and we start throwing a bunch of darts at the picture. If we count up how many darts are in the circle and divide it by how many darts are in the square then we get an approximation of the ratio!

We can simulate this in a program where we use a loop that picks random coordinates for the darts.

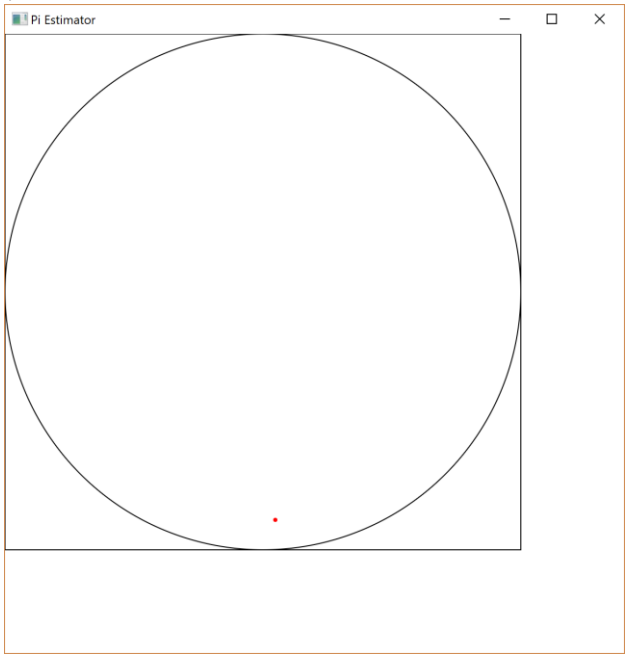
Here is a starter program that draws a circle inside a square. I made the radius 250.

```
import javafx.application.Application;
import javafx.scene.canvas.Canvas;
import javafx.scene.Scene;
import javafx.scene.Group;
import javafx.stage.Stage;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;

public class PiEstimator extends Application
{
   public static void main(String[] args)
   {
      launch(args);
   }

   @Override
   public void start(Stage primaryStage) throws Exception
   {
            final int RADIUS = 250;
            Group root = new Group();
            Scene scene = new Scene(root);
            Canvas canvas = new Canvas(600, 600);
            GraphicsContext gc = canvas.getGraphicsContext2D();

            gc.strokeRect(0,0,RADIUS*2,RADIUS*2);
            gc.strokeOval(0,0,RADIUS*2,RADIUS*2);

            root.getChildren().add(canvas);
            primaryStage.setTitle("Pi Estimator");
            primaryStage.setScene(scene);
            primaryStage.show();
   }
}
```

Here we throw a single “dart” somewhere inside the square and make it red if it’s in the circle and black otherwise:

```
import javafx.application.Application;
import javafx.scene.canvas.Canvas;
import javafx.scene.Scene;
import javafx.scene.Group;
import javafx.stage.Stage;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;
import java.util.Random;

public class PiEstimator extends Application
{
   public static void main(String[] args)
   {
      launch(args);
   }
```

```
    @Override
    public void start(Stage primaryStage) throws Exception
    {
            final int RADIUS = 250;
            Random rnd = new Random();
            Group root = new Group();
            Scene scene = new Scene(root);
            Canvas canvas = new Canvas(600, 600);
            GraphicsContext gc = canvas.getGraphicsContext2D();

            gc.strokeRect(0,0,RADIUS*2,RADIUS*2);
            gc.strokeOval(0,0,RADIUS*2,RADIUS*2);

            int dartX = rnd.nextInt(RADIUS*2);
            int dartY = rnd.nextInt(RADIUS*2);
            double distance = Math.sqrt((dartX-RADIUS)*(dartX-RADIUS) +
                                        (dartY-RADIUS)*(dartY-RADIUS));
            if (distance < RADIUS)
            {
                  gc.setFill(Color.RED);
            }
            else
            {
                  gc.setFill(Color.BLACK);
            }
            gc.fillOval(dartX-2,dartY-2,4,4);

            root.getChildren().add(canvas);
            primaryStage.setTitle("Pi Estimator");
            primaryStage.setScene(scene);
            primaryStage.show();
    }
}
```



Now all we need to do is put this in a loop, count how many land inside the circle vs. the square, and compute our ratio!

```
int darts = 0;
int landInCircle = 0;
while (darts < 1000)
{
      int dartX = rnd.nextInt(RADIUS*2);
      int dartY = rnd.nextInt(RADIUS*2);
      double distance = Math.sqrt((dartX-RADIUS)*(dartX-RADIUS) +
                              (dartY-RADIUS)*(dartY-RADIUS));
      if (distance < RADIUS)
      {
            gc.setFill(Color.RED);
            landInCircle++;
      }
      else
      {
            gc.setFill(Color.BLACK);
      }
      gc.fillOval(dartX-2,dartY-2,4,4);
      darts++;
}
double ratio = 4.0 * landInCircle / darts;
// All darts thrown and in Square
System.out.println("The estimate of PI is " + ratio);
```

If we make the circle bigger and throw more darts we can get a better estimate. Note there are better ways to estimate PI but this is a good visual depiction on how to compute it!

Example: What is the output of this code?

```
int j=0, k=0, x=5;
while (j<x)
{
      System.out.print("*");
      j++;
}
System.out.println();       // Prints a newline
j=0;
while (j<x-2)
{
      System.out.print("*");
      k=0;
      while (k<x-2)
      {
            System.out.print(".");
            k++;
      }
      System.out.println("*");
      j++;
}
j=0;
while (j<x)
{
      System.out.print("*");
      j++;
}
System.out.println();
```

This last example illustrates the concept of nested loops. It is possible, and often desirable, to insert one loop inside another loop. When this is done, it is called a nested loop.

Do-While Loop

It turns out that we can do all of the looping we need with the while loop. However, there are a number of other looping constructs make it easier to write certain kinds of loops than others. Consider the do-while loop, which has the following format:

```
do {
        statement1;
        statement2;
        …
        statement N;
} while (Boolean_condition);
```

The do-while loop executes all of the statements as long as the Boolean condition is true. How is this different from the while-do loop? In the do-while loop, the computer **always** executes the body of the loop at least once before it checks the Boolean condition. In the while-do loop, the Boolean condition is checked **first**. If it is false, then the loop's body is never executed.

For example, we could rewrite the do-while loop using an if-statement and a while loop:

```
if (Boolean_condition) {
        do {
                statement1;
                statement2;
                …
                statement N;
        } while (Boolean_condition);
}
```

This would be equivalent to the while loop.

As an example, let's convert the while loop we wrote to input numbers into a do-while loop. In the original example, we had to add an if-statement to check for the sentinel value inside the loop (because we don't want to add –9999 to the sum). This can be rewritten using a do-while loop without requiring the if-statement:

```
      Scanner keyboard = new Scanner(System.in);
      String s;
      int sum=0, x=0;
      do
      {
            sum = sum + x;
            System.out.println(“Enter an integer:”);
            x = keyboard.nextInt();
      } while (x!=-9999);
      System.out.println("The sum of the numbers you entered is "
+ sum);
```

Since the input comes at the end of the loop after the sum, we won't be adding in –9999. If the user types –9999, the loop will exit. Note that the first time through the loop, sum will get set to sum + x. However, by initializing x to zero, sum remains the same at 0, so if the first value typed is –9999, we will still get sum=0 in both cases.

Another place where a do-while loop is useful is to print menus and check for valid input:

```
      int i;
      Scanner keyboard = new Scanner(System.in);
      String s;
      System.out.println("Main Menu.  Enter 1 to perform task
one, or 2 to perform task two.");
      do
      {
            System.out.println("Enter choice:");
            i = keyboard.nextInt( );
      } while ((i != 1) && (i !=2 ));
```

This loop will continue as long as the user types in something that is neither '1' nor '2'.

Exercise: What would happen if the loop was:

```
      do {
            System.out.println("Enter choice:");
            i = keyboard.nextInt( );
      } while ((i!=1) || (i!=2));
```

The For Loop

The for loop is a compact way to initialize variables, execute the body of a loop, and change the contents of variables. It consists of three expressions that are separated by semicolons and enclosed within parentheses:

```
for (expression1; expression2; expression3)
        statement;
```

Where, statement might include a { block } of statements:

```
for (expression1; expression2; expression3) {
        Statement;
        …
}
```

All of the expression statements are optional!

Expression1 is used to set initial values, and can set multiple values separated by a comma.

Expression2 is the condition for the loop to continue (while this is true).

Expression3 contains any operations you'd like to do at the end of each iteration. Separate different instructions with a comma.

The for loop can be written in the following equivalent while-loop format:

```
expression1;
while (expression2) {
        statement;  …
        expression3;
}
```

Here are some typical uses of for-loops:

```
int sum, i, value;
Scanner keyboard = new Scanner(System.in);
String s;
for (i=0, sum=0; i<10; i++) {
      System.out.println("Enter an integer:");
      value = keyboard.nextInt();
      sum = sum + value;
}
```

This snippet loops from 0 to 9 and keeps a sum of values input by the user and saved in the variable sum.

We could do the same thing but count backwards from 10:

```
for (i=10, sum=0; i>0; i--) {
     System.out.println("Enter an integer:");
     value = keyboard.nextInt();
     sum = sum + value;
}
```

This loop ends when i=0.

Note that sometimes we can use a for loop to do work without any body at all!

```
for (i=2; i<=1000; i=i*2);
```

This snippet of code produces the first power of 2 larger than 1000. Note where the semicolon is placed to avoid any body at all. All the work is done in the loop heading.

However, normally when there is a semicolon at the end of the loop statement, it is a bug. Consider the following:

```
int product=1, num;
for (num = 1; num <=5; num++);
     product = product * num;
System.out.println(num);
```

You might expect this to print out 5*4*3*2*1, or 120. Instead, it prints out 6. Why? Because of the semicolon at the end of the for loop:

```
for (num = 1; num <=5; num++);
```

The semicolon terminates the loop, so there is no loop body executed for each iteration. Product is then set to product * num, which is now 6. To fix this bug, we should remove the semicolon from the for loop statement:

```
int product=1, num;
for (num = 1; num <=5; num++)
     product = product * num;
System.out.println(num);
```

Here is an example showing that the expressions in the loop header are optional. The following is equivalent to the previous example:

```
i=10; sum=0;
for (;i>0;) {
      System.out.println("Enter an integer:");
      value = keyboard.nextInt();
      sum += value;
      i--;
}
```

This is equivalent to the previous example we did with the initialization and loop decrement all contained within the loop header.

Finally, consider the following:

```
for (;;) {
      System.out.println("hi");
}
```

This is equivalent to an infinite loop. We will print out "hi" forever until the user stops it by hitting control-c or stopping it from an IDE environment.

Finally, it is common to declare a variable right at the beginning of the for loop. If we do this then the variable is considered local to the for loop. It only exists inside the for loop. If we try to access the variable outside the for loop, it is illegal. This technique is normally used for a loop index that only applies to the loop, not outside.

```
for (int i = 0; i < 10; i++)
{
      System.out.println(i);  // Can use i inside the loop
}
// If we try to access i here, it is a syntax error
```

Break and Continue

Let's expand on the basic while loop a bit by introducing the break and continue statements. Of these operations, break is the more commonly seen instruction. Both break and continue are statements that alter the flow of execution within a control structure. Break is used with the Switch statement, the While statement, the Do-While statement, and the For statement. (we will define the switch statement below.) Break interrupts the flow of control by immediately exiting while within these statements. In contrast, continue is used only with looping statements. It alters the flow of control by immediately terminating the current iteration. Note the difference between continue and break in a loop: continue skips to the next iteration of the loop, and break skips to the statement following the loop.

Break:

```
while (b) {
      first_statement;  // executed first
      break;            // skips to outside loop
      skipped_statement;
}
next_statement;         // executed second
```

Continue:

```
while (b) {
      first_statement;  // executed first
      continue;         // skips directly to end of the loop
      skipped_statement;
}
outside_statement;      // executed when b becomes false
```

Break is extremely useful with the Switch statement but should be used with caution with looping statements. Good style dictates that loops have only one entry and one exit except under very unusual circumstances.

Here is an example of break:

```
int i=1;
while (true) {
      i *= 2;
      if (i>31) break;
}
System.out.println(i);
```

This loop would normally go forever, but thanks to the break condition it will stop when i=32.

Here is an example of continue:

```
int i=0, sum=0;
while (i<10) {
      i++;
      if ((i % 2)==0) continue;
      sum += i;
}
System.out.println("Sum of odd numbers is " + sum);
```

This only adds the odd numbers, because the continue statement skips the line “sum+=i" for even numbers. The output is then 25.

As you should be able to see, typically the break and continue statements can be avoided and replaced with an if-then-else statement to make it easier to follow the program’s

execution. The only real excuse to use the continue statement is to avoid adding an extra if-statement to encapsulate your data (e.g., you might already have a lot of nested if-statements and you want to avoid another).

Exercise: Use a loop to simulate the Monty Hall problem 1000 times and output how many times you win switching and how many times you win not-switching. Use a random number for the initial pick.

Exercise: Write a program that finds and prints all of the prime numbers between 3 and 100. A prime number is a number such that one and itself are the only numbers that evenly divide it (e.g., 3,5,7,11,13,17, …)

Here is some pseudocode:

```
Loop from i=3 to 100
        Set flag = IsAPrime
        Loop from j=2 to i-1        (could we loop to a smaller number?)
                If i divided by j has no remainder
                        Then j evenly divides into i and i is not a prime number
                                Set flag = IsNotAPrime
        If flag still equals IsAPrime then
                Print i "is a prime number".
```