Lesson 30…..Random Numbers

Why random?

Why would we want random numbers? What possible use could there be for the generation of **unpredictable** numbers? It turns out there are plenty of applications, and the following list suggests just a few:

1. Predictions for life expectancy …used in insurance
2. Business simulations
3. Games …gives users a different experience each time
4. Simulations for scientific research, etc.

Important methods:

The *Random* class (requires the import of *java.util.Random*) generates random numbers and has three methods, besides the constructor, that are of interest to us. These are not *static* methods, so we must first create an object:

Constructor

public Random( ) // **Signature**

Example:
Random rndm = new Random( );

nextInt( )

public int nextInt( ) // **Signature**

This yields a randomly selected integer in the range Integer.MIN_VALUE to Integer.MAX_VALUE. (-2,147,843,648 to 2,147,843,647 as specified in Appendix C).

Example:
int x = rndm.nextInt( ); //x could be any integer from -2,147,843,648 to
//2,147,843,647

nextInt(n)

public int nextInt(int n) // **Signature**

This yields a randomly selected integer (0, 1, 2, …, n-1).

Example:
int x = rndm.nextInt(21); //x could be any integer from 0 to 20, inclusive for both

nextDouble( )

public double nextDouble( ) // **Signature**

This yields a randomly selected *double* from 0 (inclusive) to 1 (exclusive) and behaves exactly as does Math.random( ) (discussed in Lesson 6).

Example:
double d = rndm.nextDouble( ); //generates doubles in the range $0 \leq d < 1$

Because of the two versions of *nextInt*, we notice that our *Random* class has two methods of the same name (but different parameters). We say the methods named *nextInt* are

overloaded. In some contexts overloading is bad (example, overloading a truck). However, in the software sense of overloading, it is perfectly normal and acceptable.

Typical Problems:

1. Suppose we want a range of integers from 90 to 110, inclusive for both.

   First we subtract (110 – 90 = 20). Then add 1 to get 21. Now set up your code as follows to generate the desired range of integers:

```
int r = 90 + rndm.nextInt(21);
```

   Put this last line of code in a *for*-loop, and you will see a range of integers from 90 to 110. Loop through 1000 times, and likely you will see every value…most will be repeated several times.

```
int r = 0, count = 0;
Random rndm = new Random( );
for(int j = 0; j < 1000; j++)  {
        r = 90 + rndm.nextInt(21);
        System.out.print(r + "   ");

        //For convenience in viewing on a console screen, the following loop
        //produces a new line after 15 numbers are printed side-by-side.
        count++;
        if(count >15) {
                System.out.println("  ");
                count = 0;
        }
}
```

2. Suppose we wish to generate a continuous range of floating point numbers from 34.7838 (inclusive) to 187.056 (exclusive). How would we do this?
   First, subtract (187.056 – 34.7838 = 152.2722). Now set up your code as follows to generate the desired range.

```
Random rndm = new Random( );
double r;
r = 34.7838 + 152.2722 * rndm.nextDouble( );
// Generates continuous floating point numbers in the range
// 34.7838   ≤  r  <  187.056
```

Some additional methods of the *Random* class:

nextBoolean( ) … returns a random *boolean* value (*true* or *false*).

nextGaussian( ) … returns a Gaussian ("normally") distributed *double* with a mean value of 0.0 and a standard deviation of 1.0.

Project… Generate Random Integers

As described in problem 1 above, generate 33 random integers in the inclusive range from 71 to 99.

Project… Generate Random Doubles

As described in problem 2 above, generate 27 random *double*s in the inclusive range from 99.78 to 147.22.

Exercise on Lesson 30

In the following problems assume that *rndm* is an object created with the *Random* class. Assume *d* is of type *double* and that *j* is of type *int*.

1. What range of random numbers will this generate?
   j = 201 + rndm.nextInt(46);

2. What range of random numbers will this generate?
   d = 11 + 82.9 * rndm.nextDouble( );

3. What range of random numbers does *nextDouble( )* generate?

4. List all numbers that rnd*m.nextInt(10)* might generate.

5. Write code that will create an object called *rd* from the *Random* class.

6. Write code that will create a *Random* object and then use it to generate and print 20 floating point numbers in the continuous range $22.5 \leq r < 32.5$

7. What import is necessary for the *Random* class?

8. Write code that will randomly generate numbers from the following set. Printout 10 such numbers.
   18, 19, 20, 21, 22, 23, 24, 25

9. Write code that will randomly generate and print 12 numbers from the following set.
   100, 125, 150, 175

10. Write a line of code to create a *Random* class object even though *Random* wasn't imported.

Random Numbers… Contest Type Problems

1. Which of the following is a possible output?

    A. 0
    B. 36
    C. 37
    D. Throws an exception
    E. None of these

```
Random rd = new Random( );
System.out.println( rd.nextInt(36) );
```

2. To simulate the result of rolling a normal 6-sided die, what should replace **<*1>**

    A. rdm.nextDouble(6);
    B. rdm.nextInt(7);
    C. 1+ rdm.nextDouble(7);
    D. 1 + rdm.nextInt(6);
    E. 1 + rdm.nextDouble(6)

```
public static int dieOutcome( )
{
        Random rdm = new Random( );
        int die = <*1>
        return die;
}
```

3. Which of the following is a possible output of the code to the right?

    A. 0
    B. .9999
    C. 5.0
    D. 6.0
    E. None of these

```
java.util.Random rd = new java.util.Random( );
System.out.println( 1+ 5 * rd.nextDouble( ) );
```

4. What would be the range of possible values of *db* for the following line of code?
        double db = genRndDbl(4, 1);

    A. $1 \leq db < 5$
    B. $0 \leq db < 5$
    C. $1 \leq db < 4$
    D. $1 \leq db \leq 5$
    E. $0 \leq db \leq 5$

```
public static double genRndDbl(int m, int a)
{
    Random r = new Random( );
    double d = a + m * r.nextDouble( );
    return d;
}
```

5. What would be the replacement code for **<*1>** to generate random numbers from the following set?
        {20, 35, 50, 65}

    A. 20 * 15 + ri.nextInt(4);
    B. 20 + 15 * ri.nextInt(5);
    C. 15 * 20 + ri.nextInt(4);
    D. 15 + 20 * ri.nextInt(5);
    E. None of these

```
Random ri = new Random( );
int ri = <*1>
```

6. When a class has more than one method of the same name, this is called which of the following?

    A. overloading   B. inheritance   C. overriding   D. polymorphism

    E. None of these

7. Which of the following "tosses" a *Coin* object named *theCoin*, and produces a *true* when the *toss( )* method yields a *HEADS*?

A. theCoin.toss = = HEADS
B. toss = = 0
C. theCoin.toss( ) = = Coin.HEADS
D. theCoin.HEADS = = HEADS
E. Both C and D

8. Assuming that the *Random* class is "perfect" and generates all of the integers with equal probability, what is the probability that *toss( )* returns a head?

A. slightly over .5
B. slightly under .5
C. 1
D. exactly .5
E. None of these

```
public class Coin
{
    public Coin( )
    {
        r = new Random( );
    }

    public int toss( )
    {
        int i = r.nextInt( );
        if(i < 0)
        {
            return TAILS;
        }
        else
        {
            return HEADS;
        }
    }

    public static final int HEADS =  0;
    public static final int TAILS = 1;

    private Random r;
}
```

Project… Monte Carlo Technique

Imagine a giant square painted outdoors, on the ground, with a painted circle inscribed in it. Next, image that it's raining and that we have the ability to monitor every raindrop that hits inside the square. Some of those raindrops will also fall inside the circle, and a few will fall in the corners and be inside the square, but not inside the circle. Keep a tally of the raindrops that hit inside the square (*sqrCount*) and those that also hit inside the circle (*cirCount*).

The ratio of these two counts should equal the ratio of the areas as follows: (Understanding this statement is essential. It is the very premise of this problem.)

sqrCount / cirCount = (Area of square) / (Area of circle)

sqrCount / cirCount = $side^2 / (\pi * r^2)$

Solving for π from this equation we get

π = cirCount * $(side^2)$ / (sqrCount * r^2)

So why did we solve for π? We already know that it's $\cong$ 3.14159. We simply want to illustrate that by a simulation (raindrop positions) we can solve for various things, in this case something we already know. The fact that we already know π just makes it that much easier to check our answer and verify the technique.

We are going to build a class called *MonteCarlo* in which the constructor will establish the size and position of our square and circle. Public state variables inside this class will be *h*, *k*, and *r*. These are enough to specify the position and size of our circle and square as shown in the figure to the right.

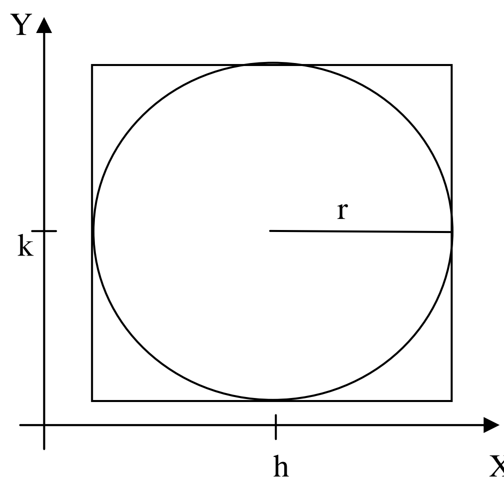

Fig. 30-1

The requirements of your *MonteCarlo* class are:

1. The constructor should receive *h*, *k*, and *r* as described above and use them to set the instance fields (state variables).

2. State variables *h*, *k*, and *r* are public *double*s. Create a *private* instance field as an object of the *Random* class. Call it *rndm*.

3. The *nextRainDrop_x( )* method should return a *double* that corresponds to a random raindrop's *x* position. The range of *x* values should be confined to the square shown above. No parameters are passed to this method.

4. The *nextRainDrop_y( )* method should return a *double* that corresponds to a random raindrop's *y* position. The range of *y* values should be confined to the square shown above. No parameters are passed to this method.

5. The method *insideCircle(double x, double y)* returns a *boolean*. A *true* is returned if the parameters *x* and *y* that are passed are either inside or on the circle.

   In writing this method, you must remember that the equation of a circle is
   $(x - h)^2 + (y - k)^2 = r^2$ …where *(h,k)* is the center and *r* is the radius.

   Also, the test for a point *(x, y)* being either inside or on a circle is
   $(x - h)^2 + (y - k)^2 <= r^2$

You will need to build a *Tester* class with the following features:

1. Class name, *Tester*

2. There is only one method, the *main( )* method.

3. Create a *MonteCarlo* object called *mcObj* in which the center of the circle is at (5, 3) and the radius is 2.

4. Set up a *for*-loop for 100 iterations:

5. Inside the loop obtain the random coordinates of a rain drop using the *nextRainDrop_x( )* and *nextRainDrop_y( )* methods.

6. Using the *x* and *y* just obtained, pass them as arguments to the *insideCircle( )* method to decide if our "raindrop" is inside the circle. If *insideCircle( )* returns a *true* then increment *cirCount*.

7. Increment *sqrCount* on each pass through the loop.

8. After the loop, calculate and print your estimate for π according to the solution for π on the previous page.

9. Change the number of iterations of the loop to 1000 and run the program again. Repeat for 10,000, 100,000, and 1,000,000 iterations. The estimate for π should improve as the number of iterations increases.