Iteration

Chapter 6

While Loops

- Executes a block of code repeatedly
- A condition controls how often the loop is executed

```
while (condition)
    statement
```

- Most commonly, the statement is a block statement (set of statements delimited by `{ }`)

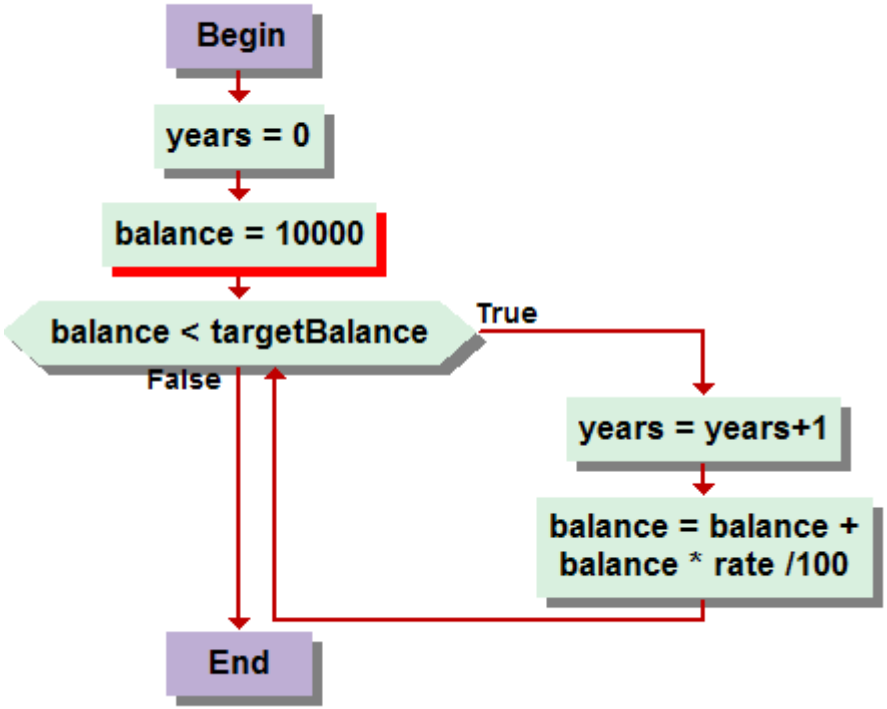
Calculating the Growth of an Investment

Invest $10,000, 5% interest, compounded annually

Year	Balance
0	$10,000
1	$10,500
2	$11,025
3	$11,576.25
4	$12,155.06
5	$12,762.82

Calculating the Growth of an Investment (Visual Logic)

- When has the bank account reached a particular balance?



Calculating the Growth of an Investment

- When has the bank account reached a particular balance?

```
int years;

while (balance < targetBalance)
{
   years++;
   double interest = balance * rate / 100;
   balance = balance + interest;
}
```

Investment.java

```
01: /**
02:    A class to monitor the growth of an investment that
03:    accumulates interest at a fixed annual rate.
04: */
05: public class Investment
06: {
07:    /**
08:       Constructs an Investment object from a starting balance and
09:       interest rate.
10:       @param aBalance the starting balance
11:       @param aRate the interest rate in percent
12:    */
13:    public Investment(double aBalance, double aRate)
14:    {
15:       balance = aBalance;
16:       rate = aRate;
17:       years = 0;
18:    }
19:
```

Investment.java (cont.)

```
20:    /**
21:       Keeps accumulating interest until a target balance has
22:       been reached.
23:       @param targetBalance the desired balance
24:    */
25:    public void waitForBalance(double targetBalance)
26:    {
27:       while (balance < targetBalance)
28:       {
29:          years++;
30:          double interest = balance * rate / 100;
31:          balance = balance + interest;
32:       }
33:    }
34:
```

Investment.java (cont.)

```
35:    /**
36:       Gets the current investment balance.
37:       @return the current balance
38:    */
39:    public double getBalance()
40:    {
41:       return balance;
42:    }
43:
44:    /**
45:       Gets the number of years this investment has accumulated
46:       interest.
47:       @return the number of years since the start of the investment
48:    */
49:    public int getYears()
50:    {
51:       return years;
52:    }
53:
54:    private double balance;
55:    private double rate;
56:    private int years;
57: }
```

InvestmentRunner.java

```
01: /**
02:    This program computes how long it takes for an investment
03:    to double.
04: */
05: public class InvestmentRunner
06: {
07:    public static void main(String[] args)
08:    {
09:       final double INITIAL_BALANCE = 10000;
10:       final double RATE = 5;
11:       Investment invest = new Investment(INITIAL_BALANCE, RATE);
12:       invest.waitForBalance(2 * INITIAL_BALANCE);
13:       int years = invest.getYears();
14:       System.out.println("The investment doubled after "
15:             + years + " years");
16:    }
17: }
```

InvestmentRunner.java (cont.)

Output:

```
The investment doubled after 15 years
```

Self Check

What would happen if `RATE` was set to `0` in the `main` method of the `InvestmentRunner` program?

Common Error: Infinite Loops

- ```
  int years = 0;
  while (years < 20)
  {
     double interest = balance * rate / 100;
     balance = balance + interest;
  }
  ```

- ```
  int years = 20;
  while (years > 0)
  {
     years++; // Oops, should have been years-
     double interest = balance * rate / 100;
     balance = balance + interest;
  }
  ```

- Loops run forever – must kill program

Common Error: Off-by-One Errors

```
int years = 0;
while (balance < 2 * initialBalance)
{
   years++;
   double interest = balance * rate / 100;
   balance = balance + interest;
}
System.out.println("The investment reached the target
   after " + years + " years.");
```

Should `years` start at 0 or 1?

Should the test be < or <=?

Avoiding Off-by-One Error

- Look at a scenario with simple values:
  initial `balance: $100`
  interest `rate: 50%`
  after year 1, the `balance` is $150
  after year 2 it is $225, or over $200
  so the investment doubled after 2 years
  the loop executed two times, incrementing `years` each time
  Therefore: `years` must start at `0`, not at `1`.

- interest rate: 100%
  after one year: `balance` is 2 * `initialBalance`
  loop should stop
  Therefore: must use <

- Think, don't compile and try at random

do Loops

- Executes loop body at least once:

```
do
   statement
while (condition);
```

- Example: Validate input

```
double value;
do
{
   System.out.print("Please enter a positive number: ");
   value = in.nextDouble();
}
 while (value <= 0);
```

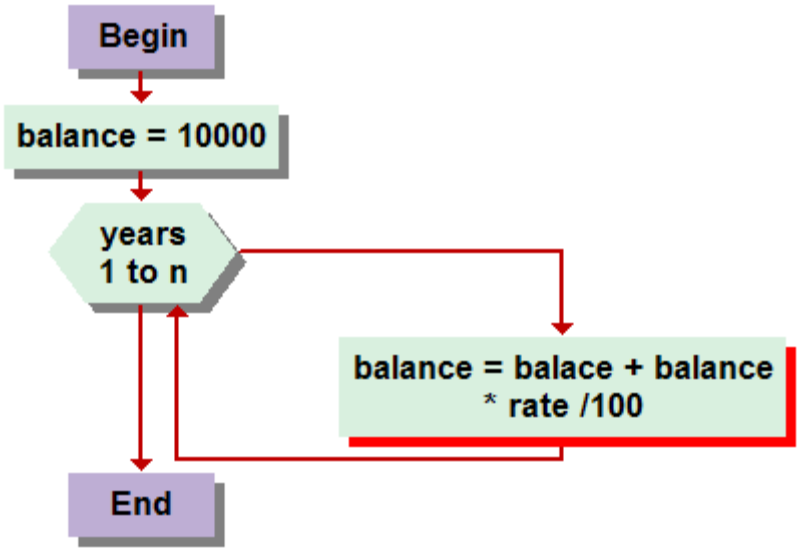
Continued

do Loops (cont.)

- Alternative:

```
boolean done = false;
while (!done)
{
   System.out.print("Please enter a positive number: ");
   value = in.nextDouble();
   if (value > 0) done = true;
}
```

for Loops


Begin
balance = 10000
years
1 to n
balance = balace + balance
* rate /100
End

for Loops (cont.)

- `for (`*`initialization`*`; `*`condition`*`; `*`update`*`)`

  `    `*`statement`*

- Example:

```
for (int i = 1; i <= n; i++)
{
   double interest = balance * rate / 100;
   balance = balance + interest;
}
```

- Equivalent to

```
initialization;
while (condition)
{ statement;
  update; }
```

Continued

for Loops (cont.)

- Other examples:

```
for (years = n; years > 0; years--) . . .
for (x = -10; x <= 10; x = x + 0.5) . . .
```

Investment.java (cont.)

```
01: /**
02:    A class to monitor the growth of an investment that
03:    accumulates interest at a fixed annual rate
04: */
05: public class Investment
06: {
07:    /**
08:       Constructs an Investment object from a starting balance and
09:       interest rate.
10:       @param aBalance the starting balance
11:       @param aRate the interest rate in percent
12:    */
13:    public Investment(double aBalance, double aRate)
14:    {
15:       balance = aBalance;
16:       rate = aRate;
17:       years = 0;
18:    }
20:    /**
21:       Keeps accumulating interest until a target balance has
22:       been reached.
```

Investment.java (cont.)

```
23:       @param targetBalance the desired balance
24:    */
26:    {
27:       while (balance < targetBalance)
28:       {
29:          years++;
30:          double interest = balance * rate / 100;
31:          balance = balance + interest;
32:       }
33:    }
34:
35:    /**
36:       Keeps accumulating interest for a given number of years.
37:       @param n the number of years
38:    */
39:    public void waitYears(int n)
40:    {
41:       for (int i = 1; i <= n; i++)
42:       {
43:          double interest = balance * rate / 100;
44:          balance = balance + interest;
```

Investment.java (cont.)

```
45:          }
46:          years = years + n;
47:       }
48:
49:       /**
50:          Gets the current investment balance.
51:          @return the current balance
52:       */
53:       public double getBalance()
54:       {
55:          return balance;
56:       }
57:
58:       /**
59:          Gets the number of years this investment has accumulated
60:          interest.
61:          @return the number of years since the start of the investment
62:       */
63:       public int getYears()
64:       {
65:          return years;
66:       }
```

Investment.java (cont.)

```
67:
68:    private double balance;
69:    private double rate;
70:    private int years;
71: }
```

InvestmentRunner.java

```
01: /**
02:    This program computes how much an investment grows in
03:    a given number of years.
04: */
05: public class InvestmentRunner
06: {
07:    public static void main(String[] args)
08:    {
09:       final double INITIAL_BALANCE = 10000;
10:       final double RATE = 5;
11:       final int YEARS = 20;
12:       Investment invest = new Investment(INITIAL_BALANCE, RATE);
13:       invest.waitYears(YEARS);
14:       double balance = invest.getBalance();
15:       System.out.printf("The balance after %d years is %.2f\n",
16:             YEARS, balance);
17:    }
18: }
```

Output:

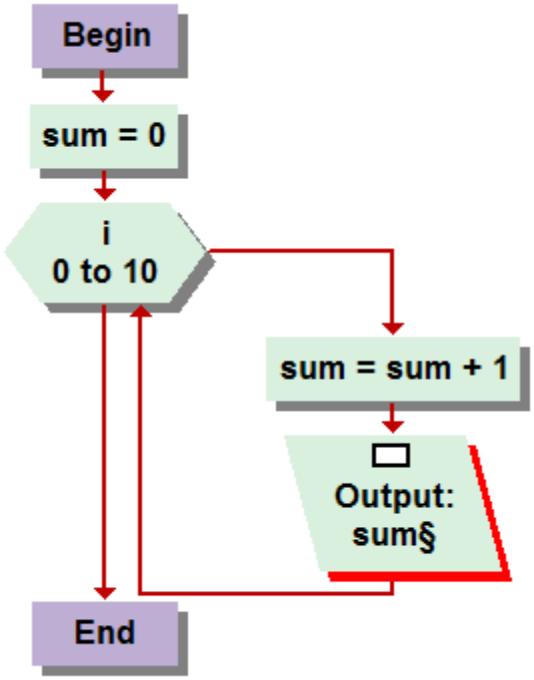
```
The balance after 20 years is 26532.98
```

Common Error

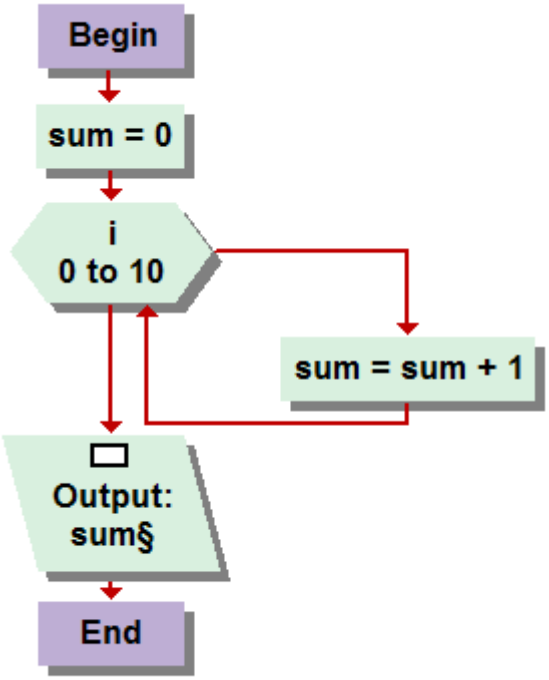
```
sum = 0;
for (int i=0; i<=10; i++);

        sum=sum+1;
        System.out.println(sum);
```

What will be printed?

Common Error in Visual Logic



Correct



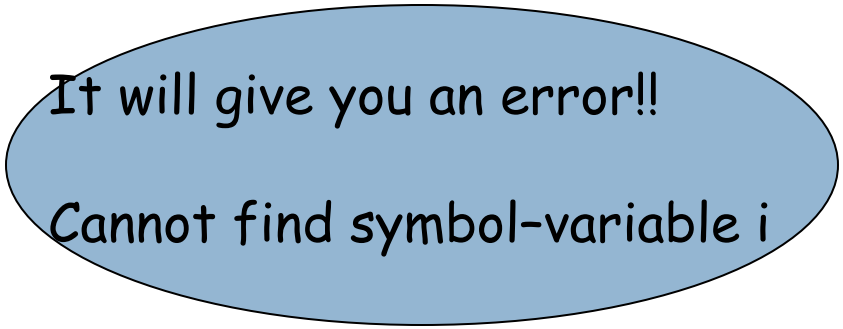
Error

Loop Variable Scope

- Scope extends to the end of the loop
- Variable is no longer defined after the loop
- If you use after the loop, you must redefine it.
- Loops can be nested
- Use different variables with each loop

Example

```
for (i=1; i<=10; i++)
        {
        for (j=1; j<=10; j++)
                {
                System.out.print(i);
                System.out.println(j);
                }
        }
System.out.println(i + " " + j);
```



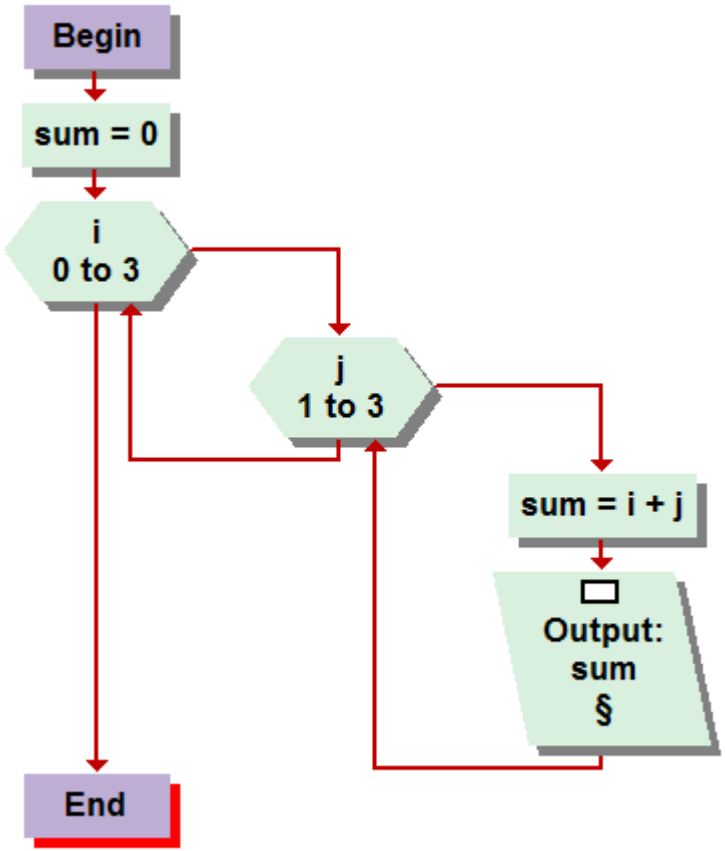
Example

```
int i = 100;
int j = 200;
for (i=1; i<=3; i++)
        {
        for (j=1; j<=3; j++)
                {
                System.out.print(i);
                System.out.println(j);
                }
        }
System.out.println(i + " " + j);
```

Output:

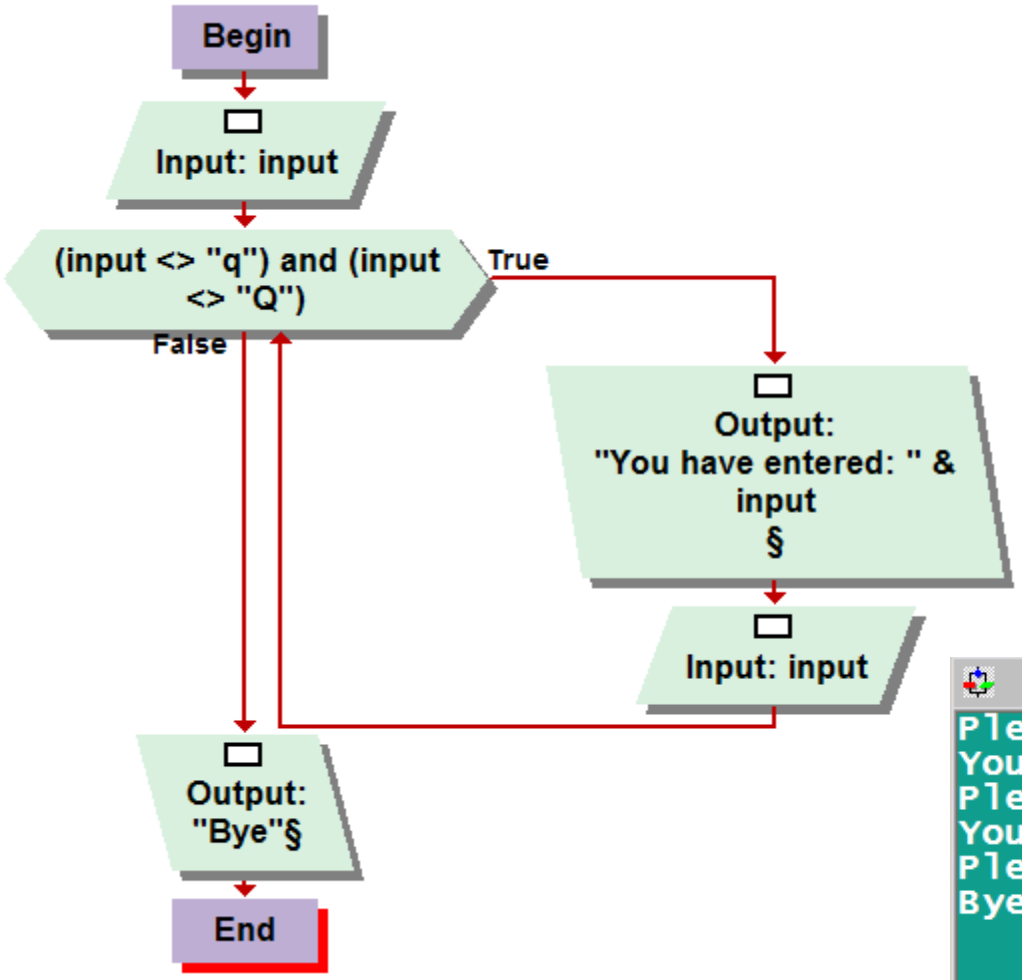
```
11
12
13
21
22
23
31
32
33
4 4
```

Nested Loop




```
int sum=0;
for (i=0; i<=3; i++)
   {
   for (j=1; j<=3; j++)
         {
         sum=i+j;
         System.out.println(sum);
         }
   }
```

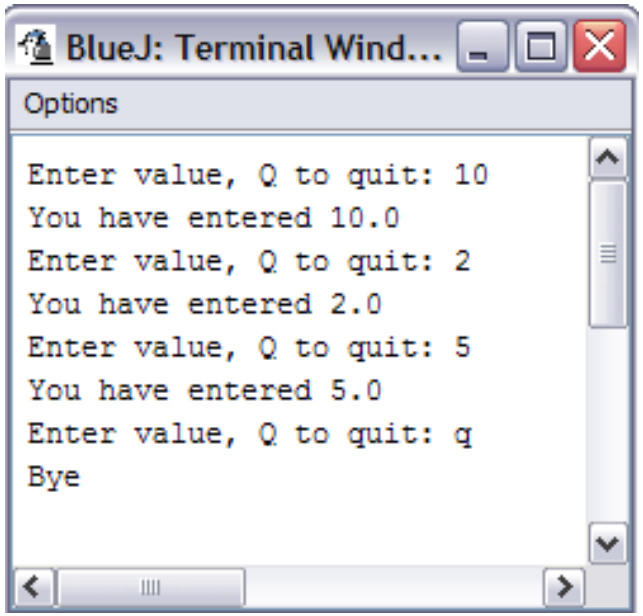
Sentinel Value




```
Please type a value for INPUT:10
You have entered: 10
Please type a value for INPUT:20
You have entered: 20
Please type a value for INPUT:"q"
Bye
```

Sentinel Value

```
System.out.print("Enter value, Q to quit: ");
Scanner in = new Scanner (System.in);
String input = in.next();
while (! input.equalsIgnoreCase("Q"))
   {
      double x = Double.parseDouble(input);
      System.out.println("You have entered " + x);
      System.out.print("Enter value, Q to quit: ");
      input = in.next();
   }
System.out.print("Bye");
```



Loop and Half

- Sometimes termination condition of a loop can only be evaluated in the middle of the loop
- Then, introduce a boolean variable to control the loop:

```
boolean done = false;
while (!done)
{
    Print prompt
    String input = read input;
    if (end of input indicated)
       done = true;
    else
    {
       Process input
    }
}
```

DataAnalyzer.java

```
01: import java.util.Scanner;
02:
03: /**
04:    This program computes the average and maximum of a set
05:    of input values.
06: */
07: public class DataAnalyzer
08: {
09:    public static void main(String[] args)
10:    {
11:       Scanner in = new Scanner(System.in);
12:       DataSet data = new DataSet();
13:
14:       boolean done = false;
15:       while (!done)
16:       {
17:          System.out.print("Enter value, Q to quit: ");
18:          String input = in.next();
19:          if (input.equalsIgnoreCase("Q"))
20:             done = true;
```

Continued

DataAnalyzer.java (cont.)

```
21:             else
22:             {
23:                double x = Double.parseDouble(input);
24:                data.add(x);
25:             }
26:          }
27:
28:          System.out.println("Average = " + data.getAverage());
29:          System.out.println("Maximum = " + data.getMaximum());
30:       }
31: }
```

DataSet.java

```
01: /**
02:    Computes the average of a set of data values.
03: */
04: public class DataSet
05: {
06:    /**
07:       Constructs an empty data set.
08:    */
09:    public DataSet()
10:    {
11:       sum = 0;
12:       count = 0;
13:       maximum = 0;
14:    }
15:
16:    /**
17:       Adds a data value to the data set
18:       @param x a data value
19:    */
20:    public void add(double x)
21:    {
```

Continued

DataSet.java (cont.)

```
22:          sum = sum + x;
23:          if (count == 0 || maximum < x) maximum = x;
24:          count++;
25:       }
26:
27:       /**
28:          Gets the average of the added data.
29:          @return the average or 0 if no data has been added
30:       */
31:       public double getAverage()
32:       {
33:          if (count == 0) return 0;
34:          else return sum / count;
35:       }
36:
37:       /**
38:          Gets the largest of the added data.
39:          @return the maximum or 0 if no data has been added
40:       */
```

Continued

DataSet.java (cont.)

```
41:    public double getMaximum()
42:    {
43:       return maximum;
44:    }
45:
46:    private double sum;
47:    private double maximum;
48:    private int count;
49: }
```

Output:

```
Enter value, Q to quit: 10
Enter value, Q to quit: 0
Enter value, Q to quit: -1
Enter value, Q to quit: Q
Average = 3.0
Maximum = 10.0
```

Random Numbers and Simulations

- In a simulation, you repeatedly generate random numbers and use them to simulate an activity
- Random number generator

```
Random generator = new Random(); int n =
generator.nextInt(a); // 0 < = n < a double x =
generator.nextDouble(); // 0 <= x < 1
```

- Throw die (random number between 1 and 6)

```
int d = 1 + generator.nextInt(6);
```

Die.java

```
01: import java.util.Random;
02:
03: /**
04:    This class models a die that, when cast, lands on a random
05:    face.
06: */
07: public class Die
08: {
09:    /**
10:       Constructs a die with a given number of sides.
11:       @param s the number of sides, e.g. 6 for a normal die
12:    */
13:    public Die(int s)
14:    {
15:       sides = s;
16:       generator = new Random();
17:    }
18:
```

Continued

Die.java (cont.)

```
19:    /**
20:       Simulates a throw of the die
21:       @return the face of the die
22:    */
23:    public int cast()
24:    {
25:       return 1 + generator.nextInt(sides);
26:    }
27:
28:    private Random generator;
29:    private int sides;
30: }
```

DieSimulator.java

```
01: /**
02:    This program simulates casting a die ten times.
03: */
04: public class DieSimulator
05: {
06:    public static void main(String[] args)
07:    {
08:       Die d = new Die(6);
09:       final int TRIES = 10;
10:       for (int i = 1; i <= TRIES; i++)
11:       {
12:          int n = d.cast();
13:          System.out.print(n + " ");
14:       }
15:       System.out.println();
16:    }
17: }
```

DieSimulator.java (cont.)

Output:

```
6 5 6 3 2 6 3 4 4 1
```

Second Run:

```
3 2 2 1 6 5 3 4 1 2
```