CS1007 lecture #8 notes

thu 26 sep 2002

- news
  - homework #2 due tue oct 1
  - homework #1 should be returned in recitation this week
  - short quiz #1 today
- the `java.util.Random` class
- the `java.util.Date` class
- introduction to recursion
- method overloading
- reading: *ch 4.7-4.13*

classes.

- *classes* are the block around which Java is organized
- classes are composed of
  - data elements:
    - *variables* — i.e., their values can change during the execution of a program
    - *constants* — i.e., their values CANNOT change during the execution of a program
      - like variables, they have a type, a name and a value
  - *methods*
    - modules that perform actions on the data elements
      - like variables, they have a type, a name and a value
      - unlike variables, the type can be *void*, which means that they don't really have a value
    - *constructors* — special types of methods used to set up an object before it is used for the first time
- groups of related classes are organized into *packages*

`java.util.Random` class (1).

- the `Random` class in the `java.util` package
- there is another way to generate random numbers besides using the `Math.random()` from the `java.lang.Math` class
- there are two methods defined in the `Random` class:

```
public Random();
public Random( long seed );
// constructor -- can be called with or without a seed

public void setSeed( long seed );
// sets the seed for the random number generator
```

- this class implements a *pseudo random number generator*
- which is really a sequence of numbers
- the *seed* tells the random number generator where to start the sequence

`java.util.Random` class (2).

- more methods defined in the `Random` class, used to get the random numbers:

```
public float nextFloat();
// returns a random number between 0.0 (inclusive) and
// 1.0 (exclusive)

public int nextInt();
// returns a random number that ranges over all possible
// int values (positive and negative)
```

`java.util.Date` class (1).

- this class is handy for getting the current date
- or creating a `Date` object set to a certain date
- some methods defined in the `Date` class:

```
public Date();
public Date( long date );
// constructor -- called without an argument, uses the
// current time; otherwise uses the time argument

public boolean after( Date arg );
public boolean before( Date arg );
public boolean equals( Object arg );
public long getTime();
public String toString();
```

- computer time is measured in milliseconds since midnight, January 1, 1970 GMT

`java.util.Date` class (2).

- a `Date` object is handy to use as a seed for a random number generator
- for example:

```
import java.util.*;
public class ex7g {
  public static void main( String[] args ) {
    Date   now = new Date();
    Random rnd = new Random( now.getTime() );
    System.out.println( "here's the first random number: "+
                        rnd.nextInt() );
  } // end of main()
} // end of class ex7g
```


methods — declaring them.

- like a variable, has:
  - data type:
    - primitive data type, or
    - class
  - name (i.e., identifier)
- also has:
  - arguments (optional)
    - also called *parameters*
    - *formal parameters* are in the blueprint, i.e., the method declaration
    - *actual parameters* are in the object, i.e., the run time instance of the class
  - throws clause (optional)
    (we'll defer discussion of this until later in the term)
  - body
  - return value (optional)

methods — using them.

- program control jumps inside the body of the method when the method is *called* (or *invoked*)
- arguments are treated like local variables and are initialized to the values of the calling arguments
- method body (i.e., statements) are executed
- method *returns* to calling location
- if method is not of type *void*, then it also *returns* a value
  - return type must be the same as the method's type
  - calling sequence (typically) sets method's return value to a (local) variable; or uses the method's return value in some way (e.g., a print statement)

object relationships.

- are hierarchical
- example:

```
java.lang.Object
  |
  +--java.lang.Number
        |
        +--java.lang.Integer
```

- *is-a* relationship
  - an object that is an instance of a class
  - an `Integer` is a `Number`, which is a `Object`
  - children *inherit* properties of their parents; formally called *inheritance*
- *has-a* relationship
  - if an object declares data whose type is also a class

method overloading.

- using the same method name with formal parameters of different types
- example:
  - `java.lang.System` has a variable called `out`
  - which is a `java.io.PrintStream`
  - whose declarations include:

```
public void println();
public void println( boolean x );
public void println( char x );
public void println( char[] x );
public void println( double x );
public void println( float x );
public void println( int x );
public void println( long x );
public void println( Object x );
public void println( String x );
```


recursion.

- recursion is defining something in terms of itself
- there are many examples in nature
- and in mathematics
- and in computer graphics, e.g., the Koch snowflake (textbook, p.485)

power function.

- *power* is defined recursively: $x^y = \begin{cases} \text{if } y == 0, & x^y = 1 \\ \text{if } y == 1, & x^y = x \\ \text{otherwise}, & x^y = x * x^{y-1} \end{cases}$

here it is in a Java method.

```
public int power ( int x, int y ) {
   if ( y == 0 ) {
     return( 1 );
   }
   else if ( y == 1 ) {
     return( x );
   }
   else {
     return( x * power( x, y-1 ));
   }
 } // end of power() method
```

- Notice that `power()` calls itself!
- You can do this with any method *except main()*
- BUT beware of infinite loops!!!
- You have to know when and how to stop the recursion — what is the *stopping* condition

let's walk through `power(2,4)`.

	call	x	y	return value
1	power(2,4)	2	4	2 * power(2,3)
2	power(2,3)	2	3	2 * power(2,2)
3	power(2,2)	2	2	2 * power(2,1)
4	power(2,1)	2	1	2

- the first is the *original call*
- followed by three *recursive calls*