CS1007 lecture #13 notes

thu 7 mar 2002

- news
- creating objects (review)
- class libraries and packages
- `java.lang.String` class
- `java.util.Random` class
- `java.util.Date` class
- conditional operator
- references
- static modifier
- two-dimensional arrays
- reading: ch 2.5-2.6, 3.5 (p130-131), 5.1-5.2, 6.4

news.

- midterm #2 changed to: TUE APRIL 9
- homework#3 due today

creating objects — review.

- a class is used to create an *object*
  - the class is a blueprint; the object is what really gets built
- there are many *native* classes that come with Java
- you can also define your own classes
  - this is like inventing your own data type!
  - you can then declare variables whose *data type* is the class you invented
- all classes are made up of *members*
  - members can be variables, constants, constructors, methods
  - methods are accessed using the *dot operator*
- a variable whose data type is a class is a *reference* to an object
  - in order to create an object, you have to declare a variable whose data type is a class
  - this allocates memory for a *reference* to the object
  - THEN you have to *instantiate* the object by calling the class's *constructor*, which allocates memory for the object

class libraries and packages (1).

- classes that are related are grouped into *packages*
- packages that are related are grouped into *libraries*
- related classes are also called *APIs* — Application Programmer Interfaces
- the `import` statement is used to help the compiler and the JVM locate classes from files and packages that are external to a class
- "external" means not in the same file — a class can access the public members of any other class, but the compiler and the JVM (for run-time) may need to know where to find the definition of the other class (if the class file isn't located in the same directory as the class that refers to it)
- the `import` statement comes at the top of the file (just after your header comment :-)
- the format is like any of the following (pick one of the three different forms shown):

```
import <package.class>;
import <package.*>;
import <class>;
```

class libraries and packages (2).

- here are some examples:

```
import java.util.Date;
import java.io.*;
import Coin;
```

- the `java.lang` package is automatically imported, so you never need to import it explicitly
- the `*` symbol is a wildcard and means: import all the classes in the specified package
- you can create your own packages, just like you can create your own classes; but that is an advanced topic which we will not cover in this course
- for this course, always place in the same directory all the classes you create that refer to each other

class libraries and packages (3).

- we've already talked about and used `java.lang` classes:
  - `java.lang.Math`
  - `java.lang.String`
- today we'll talk about more methods in `java.lang.String`
- and introduce two classes from the `java.util` package:
  - `java.util.Date`
  - `java.util.Random`

`java.lang.String` class.

- some methods defined in the `String` class:

```
public String ( String str );
public char charAt( int index );
public int compareTo( String str );
public String concat( String str );
public boolean equals( String str );
public boolean equalsIgnoreCase( String str );
public int length();
public String replace( char oldChar, char newChar );
public String substring( int offset, int endIndex );
public String toLowerCase();
public String toUpperCase();
```

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

- we've already used `Math.random()` from the `java.lang.Math` class
- 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 ex13_1 {
  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 ex13_1
```

conditional operator (1).

- syntax:

```
<var> = ( <condition> ) ? <if_true_expr> : <if_false_expr>;
```

- this is another method of branching, BUT:
  - it is an *expression*
  - it *returns* a value
  - it only goes two ways, like a simple if-else
  - you can't put complex statements in the expression clauses!
- for example:

```
// this always adds a positive number to total
total += ( num > 0 ) ? num : Math.abs( num );
```

conditional operator (2).

- here's the full example:

```
import java.util.*;
public class ex13_2 {
  public static void main( String[] args ) {
    Date   now = new Date();
    Random rnd = new Random( now.getTime() );
    int num, total = 0;
    for ( int i=0; i<10; i++ ) {
      num = rnd.nextInt() % 100;
      System.out.println( "here's the random number: "+num );
      total += ( num > 0 ) ? num : Math.abs( num );
    }
    System.out.println("total="+total );
  } // end of main()
} // end of class ex13_2
```

references (1).

- when we declare a variable whose data type is a class, we are declaring an object reference variable
- that variable *refers to* the location in the computer's memory where the actual object is being stored
- *an object reference variable and an object are two separate things*
- declaration of an object reference variable:

```
Coin x;
```

- creation of an object (also called "construction", "instantiation"):

```
x = new Coin();
```

- when an object reference variable has been declared but the object it refers to has not been created, then the object reference variable is called a *null* reference

references (2).

- for example:

```
Coin x;
x.flip();
```

- will generate an error called a `NullPointerException` because the object which `x` refers to has not been instantiated
- you can use a constant called `null` to check if an object reference variable is null
- for example:

```
Coin x;
if ( x != null ) {
  x.flip();
}
```

references (3).

- an *alias* is an object reference variable that refers to an object that was previously constructed and is already referred to by another object reference variable
- for example:

```
Coin x = new Coin();
Coin y;
y = x;
y.flip();
```

- `y` is called an "alias" of `x` (and vice versa) because they both refer to the same location in the computer's memory
- you used an alias for homework #3 without knowing it
  - the `Blackjack` class declared a global variable: `Deck deck;`
  - the `Player()` constructor passed an object reference variable: `public Player( Deck deck0 )`
  - and then aliased the global variable to point to the same location as the constructor's argument: `deck0 = deck;`

references (4).

- garbage collection is necessary when all references to an object are gone
- because when there are no object reference variables, then there is no way to know where in memory an object is located
- Java handles this for you automatically
- the JVM periodically invokes *automatic garbage collection* while it is running
- all the memory that is allocated to an application but is not being used is “restored” so that it can be re-allocated to the application later
- if you want to perform some garbage collection on a class that you create yourself, then you would write a method called `finalize()` and whenever the automatic garbage collection was invoked and cleaned up an object of your class type, then your `finalize()` method would be called

references (5).

- when you pass objects as parameters (arguments) to a method, a *reference* is passed, not the actual object
- so be careful about what changes!
- here's an example using three classes (from the book, listing 5.1-5.3):
  - `Num`
  - `ParameterTester`
  - `ex13_3`

references (6).

```
public class Num {

  private int value;

  public Num( int update ) {
    value = update;
  } // end of constructor

  public void setValue( int update ) {
    value = update;
  } // end of setValue()

  public String toString() {
    return value+"";
  } // end of toString()

} // end of Num class
```

references (7).

```
public class ParameterTester {

  public void changeValues( int f1, Num f2, Num f3 ) {
    System.out.println( "start call:\t"+
                        "f1="+f1+"\tf2="+f2+"\tf3="+f3 );
    f1 = 999;
    f2.setValue( 888 );
    f3 = new Num ( 777 );
    System.out.println( "end call:\t"+
                        "f1="+f1+"\tf2="+f2+"\tf3="+f3 );
  } // end of changeValues()

} // end of class ParameterTester
```

references (8).

```
public class ex13_3 {

  public static void main( String[] args ) {
    ParameterTester tester = new ParameterTester();
    int a1 = 111;
    Num a2 = new Num( 222 );
    Num a3 = new Num( 333 );
    System.out.println( "before call:\t"+
                        "a1="+a1+"\ta2="+a2+"\ta3="+a3 );
    tester.changeValues( a1, a2, a3 );
    System.out.println( "after  call:\t"+
                        "a1="+a1+"\ta2="+a2+"\ta3="+a3 );
  } // end of main()

} // end of class ex13_3
```

references (9).

- sample output:

```
before call:    a1=111  a2=222  a3=333
start call:     f1=111  f2=222  f3=333
end call:       f1=999  f2=888  f3=777
after  call:    a1=111  a2=888  a3=333
```

- (trace shown in book on page 229)

references (10).

- the `this` reference refers to the current object, in case there are duplicate names
- for example, in homework #3:

```
public class Player {

  Deck deck; // this one

  public Player( Deck deck ) {
    // set this one to refer to the same location
    // as the argument
    this.deck = deck;
  } // end of constructor
  .
  .
  .
} // end of Player class
```

static modifier (1).

- an object reference variable is also called an *instance variable*
- because we *instantiate* the object in order to use it
- some members in some classes are *static* which means that they don't have to be instantiated to be used
- for example, all the methods in the `java.lang.Math` class are `static`
  - you don't need to create an object reference variable whose type is `Math` in order to use the methods in the `Math` class
  - e.g., `Math.abs()`
- you use the name of the class preceding the dot operator, instead of the name of the instance variable, in order to access the static members of the class
- that is why we can use `main()` without instantiating anything

static modifier (2).

- constants, variables and methods can all be static
- (except constructors, since they are only used to instantiate, so it doesn't make sense to have a static constructor)
- typically, constants are static
- in the `Coin` class from earlier lectures:

```
public class Coin {
  public static final int HEADS=0;
  public static final int TAILS=1;
  .
  .
  .
} // end of Coin class
```

- we can now access `Coin.HEADS` and `Coin.TAILS` without instantiating and/or without referring to a specific instance variable

static modifier (3).

- but static methods can only refer to local variables or to other static members
- go back to the earlier example `ex13_3`
- if we put the `changeValues()` method inside the `ex13_3` class file, then we'd need to instantiate an instance of the `ex13_3` class in order to access that method

static modifier (4).

```
public class ex13_4 {

  public static void main( String[] args ) {
    ex13_4 tester = new ex13_4();
    int a1 = 111;
    Num a2 = new Num( 222 );
    Num a3 = new Num( 333 );
    System.out.println( "before call:\t"+
                        "a1="+a1+"\ta2="+a2+"\ta3="+a3 );
    tester.changeValues( a1, a2, a3 );
    System.out.println( "after  call:\t"+
                        "a1="+a1+"\ta2="+a2+"\ta3="+a3 );
  } // end of main()

  public void changeValues( int f1, Num f2, Num f3 ) {
    System.out.println( "start call:\t"+
                        "f1="+f1+"\tf2="+f2+"\tf3="+f3 );
```

```
    f1 = 999;
    f2.setValue( 888 );
    f3 = new Num ( 777 );
    System.out.println( "end call:\t"+
                        "f1="+f1+"\tf2="+f2+"\tf3="+f3 );
  } // end of changeValues()

} // end of class ex13_4
```

two-dimensional arrays (1).

- arrays of arrays
- `String` is (a wrapper around) an array of `char`
- `String[]` is an array of an array of `char`
- also called a two-dimensional array
- two-dimensional arrays are declared like this:
  `char[][] a2;`
- and instantiated like this (for example for a 5x5 array):
  `a2 = new char[5][5];`
- the first dimension is called *row*
- the second dimension is called *column*
- so the element in the i-th row and the j-th column is accessed like this:
  `a2[`i`][`j`]`

two-dimensional arrays (2).

```
import java.util.*;

public class ex13_5 {

  char[][] square = new char[3][3];

  public static void main( String[] args ) {
    ex13_5 a2 = new ex13_5();
    a2.init();
    a2.print();
  } // end of main()
```

two-dimensional arrays (3).

```
public void init() {
  Date now = new Date();
  Random rnd = new Random( now.getTime() );
  int num;
  for ( int i=0; i<3; i++ ) {
    for ( int j=0; j<3; j++ ) {
      num = (Math.abs( rnd.nextInt() )%26)+65;
      square[i][j] = (char)num;
    } // end for j
  } // end for i
} // end of init()
```

two-dimensional arrays (4).

```
  public void print() {
    for ( int i=0; i<3; i++ ) {
      for ( int j=0; j<3; j++ ) {
        System.out.print( square[i][j] );
      } // end for j
      System.out.println();
    } // end for i
  } // end of print()

} // end of class ex13_5
```