CS1007 lecture #13 notes

thu 17 oct 2002

- news
  - revamped assessment: check web page for new points and new schedule
- arrays of objects
- references
- comparing objects
- reading: ch 6.8-6.15

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

arrays of objects (1).

- we can have arrays of anything — i.e., other data types — like classes
- for example, we can have an array of `Coin`, using the class from last lecture
- the `Coin[]` variable contains a list of addresses
- as with `int` or `char` arrays, first you must declare and instantiate the array:

```
Coin[] pocket = new Coin[10];
```

- but because the array elements are not primitive data types, you must also instantiate each array entry:

```
for ( int i=0; i<pocket.length; i++ ) {
  pocket[i] = new Coin();
} // end for i
```

arrays of objects (2).

```
public class ex13a {
  public static void main( String[] args ) {
    final int NUMCOINS = 10;
    Coin[] pocket = new Coin[NUMCOINS];
    int headcount = 0, tailcount = 0;
    // instantiate each of the coins in the array
    for ( int i=0; i<pocket.length; i++ ) {
      pocket[i] = new Coin();
    } // end for i
    // print the array
    for ( int i=0; i<pocket.length; i++ ) {
      System.out.println( "i["+i+"]="+pocket[i] );
    } // end for i
  } // end of main()
} // end of class ex13a
```

arrays of objects (3).

```
public class Coin {
  public final int HEADS = 0;
  public final int TAILS = 1;
  private int face;
  public Coin() {
    flip();
  } // end of Coin()
  public void flip() {
    face = (int)(Math.random()*2);
  } // end of flip()
  public int getFace() {
    return face;
  } // end of getFace()
  public String toString() {
    String faceName;
    if ( face == HEADS ) {
      faceName = "heads";
    }
    else {
      faceName = "tails";
    }
    return faceName;
  } // end of toString()
} // end of class Coin
```

arrays of objects (4).

- sample output:

```
i[0]=tails
i[1]=tails
i[2]=heads
i[3]=tails
i[4]=tails
i[5]=heads
i[6]=tails
i[7]=heads
i[8]=heads
i[9]=heads
```

-
-
-
- *but why do you have to instantiate twice?*
- because when you instantiate the first time:

```
Coin[] pocket = new Coin[10];
```

  you are only allocating memory for *references* for each `Coin` array element

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();
```

references (2).

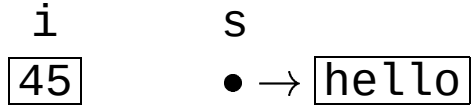
- when you declare a variable as a primitive data type, the computer sets aside a fixed amount of memory, based on the size of the data type
- when you declare a variable of any other data type (i.e., a class), you are actually declaring a *reference*
- a reference is typically the size of an *int* or a *long*
- it stores an *address* or the location in the computer's memory of where the actual data will be kept
- you can think of it like a telephone book
  - the phone book has a bunch of addresses in it
  - but not the actual buildings
  - just the *locations* of buildings

references (3).

- here's how it works inside the computer
- given the following declarations:

```
int    i = 45;
String s = "hello";
```

- the memory looks something like this:



- `i` is the label for the location in memory where the actual data is stored — in this case the `int 45`
- `s` is the label for the location in memory where the *address* is stored; the address is the location in memory where the actual data for `s` is stored
- in C, this is called a *pointer*
- we say that `s` *points to* or *references* the location in memory where the actual data for `s` is stored

references (4).

- the reference is actually a memory address, usually a `long`
- given our example on previous slide, the memory might look like this:

variable name	location in memory	value
`i`	`837542`	`45`
`s`	`837543`	**`837602`**
	`837544`	
	`837545`	
	`...`	
`s[0]`	**`837602`**	`'h'`
`s[1]`	`837603`	`'e'`
`s[2]`	`837604`	`'l'`
`s[3]`	`837605`	`'l'`
`s[4]`	`837606`	`'o'`

references (5).

- let's go back to the `Coin` example
- comment out the `toString()` method and re-run the example
- here's the output now:

```
i[0]=Coin@73d6a5
i[1]=Coin@111f71
i[2]=Coin@273d3c
i[3]=Coin@256a7c
i[4]=Coin@720eeb
i[5]=Coin@3179c3
i[6]=Coin@310d42
i[7]=Coin@5d87b2
i[8]=Coin@77d134
i[9]=Coin@47e553
```

- these are the *references* of the array elements
- we can see these reference values because we took out the `toString()` method — calling `System.out.println( pocket[i] )` automatically coerces its argument (`pocket[i]`) to a `String` so it can print it; if there is no explicit `toString()` method in the class, then a reference is the closest `String` representation

references (6).

- 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
- 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 (7).

- 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

references (8).

- 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 (9).

- 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:
  - `Num`
  - `ParameterTester`
  - `ex13b`

references (10).

```
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 (11).

```
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 (12).

```
public class ex13b {

  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 ex13b
```

references (13).

- 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
```

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
- but static methods can only refer to local variables or to other static members
- go back to the earlier example `ex13b`
- if we put the `changeValues()` method inside the `ex13b` class file, then we'd need to instantiate an instance of the `ex13b` class in order to access that method

static modifier (2).

```
public class ex13c {

  public static void main( String[] args ) {
    ex13c tester = new ex13c();
    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 ex13c
```

comparing objects (1).

- comparing two Java objects is tricky
- you have to be careful of what you are comparing:
  - is it the *value* of some member(s) of the class?
  - or is it the *reference*?
- using == compares the *references*
- which is not the same as comparing the values of member(s) of the class
- here's an example from the `Coin` class:
  - comparing the value of the `face` member of two coins:

```
if ( pocket[0].getFace() == pocket[1].getFace() ) {
  System.out.println( "coins 0 and 1 have the same face value" );
}
```

  - versus comparing the references:

```
if ( pocket[0] == pocket[1] ) {
  System.out.println( "coins 0 and 1 are the same" );
}
```

- many classes have a method called `compareTo()` to compare the value of member(s) of the class

comparing objects (2).

- in order to compare the value of two `String`s, we need to use the method
  `public int compareTo( String str )`
  from the `java.lang.String` class
- this method does a *lexical comparison* of its `String` argument with the current object (i.e., its instantiated value)
- it returns an `int` as follows:

if the current object...	*then the method returns*
is the same text as `str`	`0`
comes lexically before `str`	an `int` < 0 (e.g., -1)
comes lexically after `str`	an `int` > 0 (e.g., +1)

- using == to compare two `String`s compares their *addresses*, NOT the values of the text they store
- this is the same for comparing any two objects in Java
- most classes define a `compareTo()` method, just as most classes define a `toString()` method

comparing objects (3).

- for example:

```
public class ex13d {
  public static void main( String[] args ) {
    String s1 = new String( "hello" );
    String s2 = new String( "hello" );
    System.out.println( "s1=["+s1+"]" );
    System.out.println( "s2=["+s2+"]" );
    System.out.println( "(s1 == s2) = " + ( s1 == s2 ));
    System.out.println( "s1.compareTo(s2)="+s1.compareTo(s2));
    System.out.println( "s2.compareTo(s1)="+s2.compareTo(s1));
  } // end of main()
} // end of class ex13d
```

- sample output:

```
s1=[hello]
s2=[hello]
(s1 == s2) = false
s1.compareTo(s2)=0
s2.compareTo(s1)=0
```