CS1007 lecture #12 notes

tue 5 mar 2002

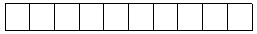
- news
- arrays
- references
- comparing `Strings`
- reading: ch 5.1, 6.1-6.2

news.

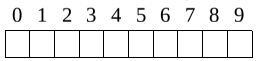
- midterm #2 changed to: TUE APRIL 9
- homework#3
  - due Thu Mar 7
  - check web page and bulletin board for hw updates

arrays (1).

- arrays are used to associate multiple instances of the same type of variable
- the `[]` indicates it's an *array*
- one example we've already used is `String[]`, which is an array of `String`
- visualize an array as a sequence of boxes, contiguous in the computer's memory, where each box stores one instance of the type of data associated with that array:



- the boxes are numbered, starting with 0 and ending with the length of the array less one; each number is called an *index*
- the *indices* for an array of 10 items can be visualized like this:



arrays (2).

- to use an array, first you must declare it:

```
int[] A;
```

- then you must instantiate it:

```
A = new int[10];
```

- or you can do both of these in one step:

```
int[] A = new int[10];
```

- then you can access its elements:

```
A[4]
```

(index=4, which is the 5th item in the array):

- you can use this accessed item just like any single data element of that type, in this case an `int`
- the number of items in the array is the variable `A.length`

arrays (3).

- here's an example that stores in an array 5 random numbers between 0 and 100:

```
public class ex12_1 {
  public static void main( String[] args ) {
    int[] A = new int[5];
    // initialize
    for ( int i=0; i<A.length; i++ ) {
      A[i] = (int)(Math.random()*100);
    } // end for i
    // print
    for ( int i=0; i<A.length; i++ ) {
      System.out.println( "i["+i+"]="+A[i] );
    } // end for i
  } // end of main()
} // end of class ex12_1
```

- sample output:

```
i[0]=12
i[1]=52
i[2]=57
i[3]=73
i[4]=67
```


arrays (4).

- 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`, 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 (5).

- here's an example:

```
public class ex12_2 {
  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 ex12_2
```


arrays (6).

```
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 (7).

- 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 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 (2).

- 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      s
45      • → hello
```

- `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 (3).

- 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 (4).

- 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

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 ex12_2 {
  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 ex12_2
```

- sample output:

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