cis20.1
design and implementation of software applications I
fall 2007
lecture # I.4

**topics:**

- introduction to java, part 3
  - static modifier
  - overriding methods
  - overloading methods
  - polymorphism and abstract classes
  - references
  - comparing objects
  - exception handling
  - arrays
  - vectors
  - streams
  - files
  - utility classes



## static modifier (1).

- when we *instantiate* an object in order to use it, we are creating an *instance variable*
  e.g., `Random r = new Random();`
- 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()`, `Math.random()`
- 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
- e.g., `Math.random()` vs `r.nextFloat()` (where `r` is the instance variable of type `Random` that we created above)
- that is why we can use `main()` without instantiating anything
  i.e., `public static void main()`



## static modifier (2).

- constants, variables and methods can all be static
- except constructors
  (since they are only used to instantiate, it doesn't make sense to have a static constructor)
- typically, *constants* are static
- example:

```
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



## overriding methods.

- when you *extend* a class, you can *override* methods defined in the parent class by defining them again in the child (and giving the child version different behavior)
- the rule is: *the version of any method that is invoked is the definition closest to the leaf of the tree*
- if you want to refer to the version of the method in a class's superclass, you use the `super` reference

## overloading methods (1).

- in addition to changing precisely what a method does, you can also change the arguments to that method
- this is very useful if you are changing the data type of data objects defined in the class
- you can create a new version of a method which has different arguments from the version of the method defined in the class's superclass
- this is what happens when we use different versions of the `println()` method:

```
int i = 5;
String s = "hello";
System.out.println( i );
System.out.println( s );
```



## overloading methods (2).

- in other words, you are 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( double x );
public void println( float x );
public void println( int x );
public void println( Object x );
public void println( String x );
```

- these are all different ways of *printing* data, but the difference is the type of *object* being printed



## other terminology...

- *polymorphism*
  - "having many forms"
  - lets us use different implementations of a single class
  - we will talked about this later in relation to *interfaces*
  - a polymorphic reference can refer to different types of objects at different times
- *abstract* class
  - represents a generic concept in a class hierarchy
  - cannot be instantiated — can only be extended



## example.

```
public class Quarter extends Coin {

  // overload constructor
  public Quarter() {
    value = 25;
    flip();
  } // end of Quarter()

  OR

  public Quarter() {
    super( 25 );
  } // end of Quarter()


} // end of class Quarter
```

## 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`
  - `ex4d`



## 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

public class ex4d {
  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 ex4d
```



## references (12).

- 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
```

## 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*? (like a pointer in C/C++)
- using == compares the *references* (addresses)
- which is not the same as comparing the values of member(s) of the class
- many classes have a method called `compareTo()` to compare the value of member(s) of the class



## comparing objects (2).

- here's an example from the `Coin` class:
  - comparing the value of the `face` member of two coins:

```
Coin coin0 = new Coin( 10 );
Coin coin1 = new Coin( 10 );
if ( coin0.getValue() == coin1.getValue() ) {
  System.out.println( "coins 0 and 1 have the same value" );
}
```

  - versus comparing the references:

```
if ( coin0 == coin1 ) {
  System.out.println( "coins 0 and 1 are the same" );
}
```



## comparing objects (3).

- in order to compare the value of two `Strings`, 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 `Strings` 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 (4).

- 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
```

## comparing objects (5).

- so we could add to our `Coin` class:

```
public int compareTo( Coin coin ) {
  if ( value == coin.getValue() ) {
    return 0;
  }
  else if ( value < coin.getValue() ) {
    return -1;
  }
  else {
    return 1;
  }
} // end of compareTo()
```



## exception handling.

- example:

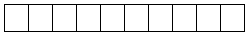
```
try {
  i = System.in.read();
}
catch ( IOException iox ) {
  System.out.println( "there was an error: " + iox );
}
```

- `try` clause contains code which may generate an exception, i.e., an error
- `catch` clause contains code to execute in case the error happens; i.e., where to go if the exception gets *caught*

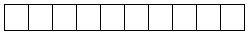


## arrays (1).

- used to associate multiple instances of the same type of variable
- the "`[]`" indicates it's an *array*
- we can have arrays of anything (i.e., other data types)
- 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 ex4a {
  public static void main( String[] args ) {
    int[] A = new int[5];
    for ( int i=0; i<A.length; i++ ) {
      A[i] = (int)(Math.random()*100);
    }
    for ( int i=0; i<A.length; i++ ) {
      System.out.println( "i["+i+"]="+A[i] );
    } // end for i
  } // end of main()
} // end of class ex4a
```



## two-dimensional arrays.

- arrays of arrays
- 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[`$i$`][`$j$`]`
-



## 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 ex4b {
  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 ex4b
```

## 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



## vectors (1).

- Java has a nice class which handles arrays dynamically: `java.util.Vector`
- the elements of a `Vector` can be any type of Java `Object`
- note that when you fetch an element from a vector, you have to cast it from a generic object to the specific class type the object should be (see example below)
- some methods:
  - constructor: `Vector();`
  - `public void addElement( Object obj );`
  - `public void insertElementAt( Object obj, int index );`
  - `public void removeElementAt( int index );`
  - `public void removeAllElements();`
  - `public void setElementAt( Object obj, int index );`
  - `public Object elementAt( int index );`
  - `public int size();`



## vectors – example.

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

public class ex4c {

  public static void main( String[] args ) {
    Vector pocket;
    int    npocket = Integer.parseInt( args[0] );

    pocket = new Vector( npocket );
    for ( int i=0; i<npocket; i++ ) {
      pocket.addElement( new Coin());
    }

    for ( int i=0; i<npocket; i++ ) {
      Coin tmp = (Coin)pocket.elementAt( i );
      System.out.print( tmp + " " );
    }
    System.out.println();
  } // end of main()

} // end of class ex4c
```

## vectors – things to notice.

- notice that we instantiate twice...
- notice that we instantiate in the call to `pocket.addElement()`:

```
pocket.addElement( new Coin());
```

- notice that we *cast* the return from `pocket.elementAt()`:

```
Coin tmp = (Coin)pocket.elementAt( i );
```



## streams (1).

- we've drawn a picture of input and output many times this semester:

  input → CPU → output

- up to now, input has been from the keyboard and output has been to the screen
- today we will read input from "text files" and write output to "text files"
- input and output flow from and to *streams*
  - a *stream* is an ordered sequence of bytes
  - streams flow from a source to a destination
  - with input, the source is the keyboard and the destination is a program
  - with output, the source is a program and the destination is the screen



## streams (2).

- thus there are two categories of streams:
  - *input streams*
  - *output streams*
- streams can also be subdivided based on their content:
  - *character streams* (i.e., text)
  - *byte streams* (i.e., binary data)
- or their usage:
  - *data streams* (e.g., String in memory, file on disk)
  - *processing streams* (manipulation of a data stream)



## streams (3).

- in order to handle streams in Java, we need several classes from the `java.io` package:
- classes that handle *byte streams*
  - `InputStream` ← `FileInputStream`
  - `OutputStream` ← `FileOutputStream` ← `PrintStream`
- classes that handle *character streams*
  - `Reader` ← `BufferedReader`
  - `Writer` ← `BufferedWriter`
- for example, in `java.lang.System`:
  - `System.in` is an `InputStream`
  - `System.out` is a `PrintStream`

## files (1).

- typically, there are three processing steps when using files:
  1. open
  2. read, write or update
  3. close
- we'll only talk about read and write in Java (not update)
- in order to implement file I/O in Java, we need several classes from the `java.io` package:
  - `FileReader`
  - `FileWriter`
  - `BufferedReader`
  - `BufferedWriter`
  - `PrintWriter`



## files – reading example.

```
import java.io.*;

public class ex4f {

  public static void main( String[] args ) {
    String line = "";
    // read data from file into program variables
    try {
      FileReader fr = new FileReader( "data.dat" );
      BufferedReader infile = new BufferedReader( fr );
      line = infile.readLine();
      infile.close();
    }
    catch( FileNotFoundException fnfx ) {
      System.out.println( "file not found: data.dat" );
    }
    catch( IOException iox ) {
      System.out.println( iox );
    }
    System.out.println( "line=["+line+"]" );
  } // end of main()

} // end of ex4f class
```



## files – writing example.

```
import java.io.*;

public class ex4g {

  public static void main( String[] args ) {
    try {
      FileWriter fw = new FileWriter( "myfile.dat" );
      PrintWriter outfile = new PrintWriter( new BufferedWriter( fw ));
      outfile.println( "hello world" );
      outfile.close();
    }
    catch( IOException iox ) {
      System.out.println( iox );
    }
  } // end of main() method

} // end of ex4g class
```



## files – using them.

- the simple model for programs that work with data files is to:
  1. open the data file for reading
  2. read the contents of the data file into program variables
  3. close the data file
  4. manipulate the values in the program variables
  5. open the data file for writing
  6. write the manipulated values to the data file
  7. close the data file
- useful input class: `StringTokenizer`
- useful output class: `DecimalFormat`

## utility classes: `java.util.StringTokenizer`.

- used to break up a string into "tokens", i.e. components
- each token is separated by a "delimiter"
- default delimiter is whitespace
- but you can set another value for delimiter
- primary method used: `public String nextToken();`
- example:

```
line = infile.readLine();
tokenizer = new StringTokenizer( line );
name = tokenizer.nextToken();
try {
  units = Integer.parseInt( tokenizer.nextToken() );
  price = Float.parseFloat( tokenizer.nextToken() );
}
catch( NumberFormatException nfx ) {
  System.out.println( "error in input; line ignored: " + line );
}
```



## utility classes: `java.text.DecimalFormat`.

- used to format decimal numbers
- construct an object that handles a format
- use that format to output decimal numbers
- formatting patterns include:
  - 0 used to indicate that a digit should be printed, or 0 if there is no digit in the number (i.e., leading and trailing zeros)
  - # used to indicate that if there is a digit in the number, then it should be printed; indicates rounding if used to the right of the decimal point
- example:

```
DecimalFormat fmt = new DecimalFormat( "#.00" );
double price;
System.out.println( "price = $" + fmt.format( price ));
```



## exercises.

- start with the example class ex4a that stores in an array 5 random numbers between 0 and 100
- write a method that finds the minimum number and returns its index;
  modify the main to call the method and print out the smallest number
- modify the main to ask the user how big she wants the array to be,
  read the user's answer from the keyboard as a String,
  convert the String to an int and
  use it as the size of the array
- write a method that writes the contents of the array to a file;
  modify the main to call this method