cs3101-003 Java: lecture #3

- news:
  - homework #2 due today
  - homework #3 out today
- today's topics:
  - classes and objects
  - formatting output
  - writing your own classes
  - making sense of keywords
    * this
    * super
    * final
    * public
    * private
    * static

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*

classes: define objects.

- are "blueprints" for creating *instances* of objects
- example: a house
  - class = architect's blueprint
  - instance = a house built following that blueprint
- *instantiate* = to build the house
- you can build MANY houses using the same blueprint, so you can instantiate many objects using the same class

classes: contain members.

- **data declarations** *(e.g., the people and the stuff inside the house)*
  - constants
  - variables
- **methods** *(e.g., the things people do with the stuff)*
  - actions that are performed on the object and/or with its data
  - a *constructor* is a special method used to *instantiate* an object of that class
  - some methods may change the values of the variables
  - some methods may *return* the values of the variables
- **scope** *(e.g., where can people do things with the stuff?)*
  - *local* vs *global*
  - *instance data*
  - *method data*

classes: instantiating objects.

- in order to use a class, you *instantiate* it by creating an *object* of that type
- this is kind of like declaring a variable

```
import java.util.*;
public class ex3a {
  public static void main( String[] args ) {
    Date   now = new Date();
    Random rnd = new Random( now.getTime() );
    System.out.println( "here are ten positive integers:" );
    for ( int i=0; i<10; i++ ) {
      System.out.println( Math.abs( rnd.nextInt() ));
  } // end of main()
} // end of class ex3a
```

writing your own classes (1).

- you can create your own classes in two ways:
  - by writing a completely new class
  - by *extending* an existing class

writing your own classes (2).

- when you write your own class, you can define
  - “global” data elements
    - variables
    - constants
  - methods
  - constructor

variables.

- have a name, type and value
- value is initialized, to 0 for numbers (unlike C)
- have “global” scope if they are declared outside of any method

constants.

- their values CANNOT change during the execution of a program
- i.e., their values remain *constant*
- like variables, they have a type, a name and a value
- the keyword `final` indicates that the variable is a *constant* and its value will not change during the execution of the program
- example:

```
public class java.lang.Math {
  static final double PI=3.1415927...;
  .
  .
  .
} // end of Math class
```

method declaration.

- 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)

method use.

- 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)

constructor.

- a constructor is a special method that is invoked when an object is *instantiated*
- a constructor can have arguments, like any other method
- a constructor does not return a value
- a constructor's name is the same as the name of the class to which it belongs
- a constructor is invoked by using the *new* keyword
- example:

```
Date now = new Date();
Random r1 = new Random();
Random r2 = new Random( now.getTime() );
```

encapsulation and visibility.

- objects should be self-contained and *self-governing*
- only methods that are part of an object should be able to change that object's data
- some data elements should not even be seen (or visible) outside the object
- *public* data elements can be seen (i.e., read) and modified (i.e., written) from outside the object
- *private* data elements can be seen (i.e., read) and modified (i.e., written) ONLY from inside the object
- typically, **variables** are **private** and **methods** that provide access to them (both read and write) are **public**
- typically, **constants** are **public**
- example: house
  - walls provide privacy for the inside
  - windows provide public viewing of some of the inside

example.

```
public class Coin {

  // declare constants
  public static final int HEADS = 0;
  public static final int TAILS = 1;

  // declare variables
  private int face;
  private int value;

  // constructor
  public Coin( int value ) {
    this.value = value;
    flip();
  } // end of Coin()
```

```
// flip the coin by randomly choosing a value for the face
public void flip() {
  face = (int)(Math.random()*2);
} // end of flip()

// return the face value
public int getFace() {
  return face;
} // end of getFace()

// return the coin's value
public int getValue() {
  return value;
} // end of getValue()
```

```
  // return the coin’s face value as a String
  public String toString() {
    String faceName;
    if ( face == HEADS ) {
      faceName = "heads";
    }
    else {
      faceName = "tails";
    }
    return faceName;
  } // end of toString()

} // end of class Coin
```

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

inheritance.

- *inheritance* is the means by which classes are created out of other classes
- it is a cornerstone of object-oriented programming
- the idea is to create classes that can be re-used from one application to another
- classes contain *data objects* and *methods*
- you want to be able to change the *data type* of the data objects and still be able to use the same methods
- you also want to be able to change the flavor of what the methods do

inheritance tree (1).

- think of the most primitive Java class, `Object` as being at the root of the inheritance tree
- all other classes are “children” or *subclasses* of that class
- here is an example of the inheritance tree for `Integer`:

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

- `Integer` is a subclass of `Number` and `Number` is a subclass of `Object`
- `Integer` is also a subclass of `Object`
- conversely a parent is also called a *superclass*
- `Object` is a superclass of `Number` and `Number` is a superclass of `Integer`
- `Object` is also a superclass of `Integer`
- `Object` is also called the *base class* of `Integer`

inheritance tree (2).

- as you move DOWN the inheritance tree from the root to the leaf, you are *extending* subclasses from parent classes
  - parent classes are also called *superclasses*
  - or *base classes*
  - children classes are *derived* from their parents
- as you move UP the inheritance tree from the leaf to the root, you can say that each subclass is a *more specific* version of its parent
- this is known as the *is-a* relationship between a subclass and the parent class that the child extends
- the keyword `this` is used to specify a member of the current or immediate class

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
```

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
- 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 `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 (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()
```

exercise.

- create a class called `Card` which is a playing card
- the card has a face (hearts, diamonds, clubs or spades)
- the card has a value (2..10, J, Q, K, A), all face cards have value 10
- define a constructor that randomly sets the card's face and value
- define methods to return the card's face and value
- define another method called `pick` that will change the card's face and value, as if you picked another card from the deck
- create a second class that contains a `main()` method
- define variable(s) in the second class of type `Card`
- loop inside the `main()`, randomly picking cards until the total is greater than or equal to 21
- assume that you replace each card in the deck immediately after it has been picked (so you don't have to keep track of which cards you have picked)
- *extension:* modify the exercise so that you do keep track of which cards have been picked