cisc3120
design and implementation of software applications I
spring 2015
lecture # I.3

topics:

- introduction to java, part 2
  - java classes
  - writing your own classes
  - static modifier
  - overloading methods
  - arrays of objects

java classes (1).

- *classes* are the block around which Java is organized
- classes are composed of
  - *fields* (data elements)
    - *variables*
    - *constants*
  - *methods*
    - modules that perform actions on the data elements
- classes are *hierarchical*
- groups of related classes are organized into *packages*

more classes (1): objects.

- Classes 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

more classes (2): contain members.

- **data elements** *fields (e.g., the people and the stuff inside the house)*
  - constants — i.e., their values CANNOT change during the execution of a program
  - variables — i.e., their values can change during the execution of a program
- **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 data elements
  - some methods may *return* the values of the data elements
- **scope** *(e.g., where can people do things with the stuff?)*
  - *local* vs *global*
  - *instance data*
  - *method data*

more classes (3): 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()));
  }
}
```


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).

- Simplest form for a class definition in Java:

```
class <classname> {
  constructor_1
  constructor_2
  ...
  method_1
  method_2
  ...
  field_1
  field_2
  ...
}
```


writing your own classes (3): 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 class
- *public* data elements can be seen (i.e., read) and modified (i.e., written) from outside the class
- *private* data elements can be seen (i.e., read) and modified (i.e., written) ONLY from inside the class
- typically, classes declare **data elements** as **private** and provide **public** *mutator* and *accessor* **methods** if necessary.
- **mutator** methods commonly have *set* prefix in their name.
- **accessor** methods commonly have *get* prefix in their name.

writing your own classes (4): variables and constants.

- have a name, type and value
- values are initialized (0 for numbers, false for boolean and `null` for object variables)
- have class scope if they are declared outside of any method
- after instantiation, constant variables CANNOT change during the execution of a program
- i.e., they cannot be assigned other values, therefore remain *constant*
- 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
```


writing your own classes (5): method declaration.

- like a variable, has:
  - (return) data type:
    - primitive data type, or
    - class
    - *void* if not returning any data
  - name (i.e., identifier)
- also has:
  - parameters (optional)
    - *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

writing your own classes (6): method use.

- program control jumps inside the body of the method when the method is *called* (or *invoked*)
- arguments are treated like local variables (call-by-value) 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
  - type of the returned value must be the same as the method's return 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)

writing your own classes (7): 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

writing your own classes (8): constructor.

- example:

```
public class Student{
  // constructor sets the student id randomly
  public Student() {
    id = Math.random() * 10000;
  }

  int id;
} // end of Student class
...
  Student s1 = new Student();   // instantiates a new Student object
```


writing your own classes (9): example.

```
public class Coin {
  // declare constants
  public static final int HEADS = 0;
  public static final int TAILS = 1;

  public Coin(int value) {
    this.value = value;
    flip();
  }

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



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

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



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

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

} // 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

static modifier (3).

```
public class Student {
  public Student(){
    id = nextId;
    nextId++;
  }
  ...
  private final int id;
  private static int nextId = 1;
} // end of Student class
```


overloading methods (1).

- in addition to changing precisely what a method does, you can also change the parameters 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 parameters 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

overloading constructor.

```
public class Coin {
  // default constructor
  public Coin() {
    value = 25;
    flip();
  }

  // overloaded constructor
  public Coin(int v) {
    value = v;
    flip();
  }
}
```


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`, objects
- 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();
}
```


arrays of objects (2).

```
public class ObjectArrayDemo {
  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();

    // print the array
    for(int i = 0; i < pocket.length; i++)
      System.out.println("i["+i+"]="+pocket[i]);
  }
}
```

arrays of objects (3).

```
public class Coin {
  public final int HEADS = 0;
  public final int TAILS = 1;

  public Coin() { flip(); }

  public void flip() { face = (int)(Math.random() * 2); }

  public int getFace() { return face; }

  public String toString() {
    String faceName;
    if ( face == HEADS ) {
      faceName = "heads";
    }
    else {
      faceName = "tails";
    }
    return faceName;
  }

  private int face;
} // 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