CS1007 lecture #12 notes

tue 15 oct 2002

- news
- objects
- classes
- constants
- methods (review)
- encapsulation and visibility
  (the `public` and `private` modifiers)
- instantiation
  (the `static` modifier)
- reading: ch 6.1-6.7

objects.

- objects have:
  - state
  - set of behaviors
- example: a robot
  - state
    - where it is
    - where it was a minute ago
    - how fast its motors are turning now
    - how fast its motors can turn
  - behaviors
    - turn
    - go forward
    - go backward
    - stop

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*

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 Coin {
  final int HEADS=0;
  final int TAILS=1;
  .
  .
  .
} // end of Coin 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)

object relationships.

- are hierarchical
- example:

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

- *is-a* relationship
  - an object that is an instance of a class
  - an `Integer` *is-a* `Number`, which *is-a* `Object`
  - children *inherit* properties of their parents; formally called *inheritance*
- *has-a* relationship
  - if an object declares data whose type is also a class

method overloading.

- 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

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

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