ICS 111
Java Classes

- a simple example
- instance methods and static methods
  - accessor and mutator instance methods
- instance variables
- modifiers: public, protected, private
- javadoc
- constructors
  - default constructors
- the `null` object reference

Example: Remembering a Value

- A simple kind of object might just remember a single value, perhaps of type String:

```
public class OneString {
  private String myString;
}
```

- if you declare two objects of type OneString, each has its own copy of myString

```
OneString a = new OneString();
OneString b = new OneString();
```

  `a.myString` may hold a different value from `b.myString`

- the variable myString is an instance variable: each instance of the class (each object of type OneString) has its own copy of myString
  - can declare a variable `static` to have a class variable, that is, one global variable:

    ```
    private static String oneStringToRuleThemAll;
    ```

  - class variables are rare,instance variables are much more useful and much more common
- Now, let’s think about what methods `OneString` might need.

Accessor Methods

- Given an object of type OneString, an obvious thing to want to do is to want to get the string stored in the object

```
OneString a = new OneString();
String s = a.get();
```

- get is called an accessor method because it gives access to the values of the instance variables

```
public class OneString {
  private String myString;
  public String get() {
    return myString;
  }
}
```

- an instance method (a method declared without `static`) can use the instance variables of a class
  - a static method such as `main` cannot access instance variables
- instance methods can only be called from an actual object:
  - `OneString.get()` would not compile
- the ArrayList.get(index) method is an accessor method

Mutator Methods

- As well as accessing (reading) the value in an object of type OneString, we may want to change the value:

```
OneString a = new OneString();
a.set(“hello world");
```

- set is called a mutator method because it modifies (mutates) the values of the instance variables

```
public class OneString {
  private String myString;
  public void set(String newValue) {
    myString = newValue;
  }
}
```

- `ArrayList.set(index, value)` is a mutator method

Instance variables are always private

- For the simple example of OneString, the instance variable *could* be public
- in more complicated classes, instance variables should be **consistent** with one another
- consistent means that changing one requires changing the other in an appropriate way
- for example, in a class keeping track of bank account values, a bank transfer requires modifying two different accounts in a consistent way
- if we allow code outside the class direct access to instance variables, it is hard to be sure that all the code accessing these variables maintains the needed consistency
- therefore, as a general rule (general for Java, not just for this class!):
  - no instance variable is ever public
    - notable exception: array.length
  - for now, this means instance variables are always declared private
  - later we will learn about protected variables
  - access to instance variables is available through accessor methods and mutator methods

Instance variables are always private: part 2

- For the simple example of OneString, there is only one obvious implementation of all the methods
- for a more complicated class, we may start with a simple implementation, and redesign it later
  - the simple implementation may just have stub methods and bare-bones variables
  - the later implementation may be full-fledged
- we want the code outside this class to only depend on the interface of the class (i.e. the method headers), not on the implementation!
  - if `String` is re-implemented, none of the code that has `String` variables or `String` values or calls `String` methods should have to change in any way!

Modifiers: public, protected, private

- A public method can be called by any code in any class in your program
- A private method can only be called from code within the same class
  - a private instance variable or a private class variable is only in scope within the class
- A protected method can be called within the class, or within any class that is below this class in the class hierarchy
  - that is, any class that **inherits** from this class
  - what does a class inherit from another class? public methods, protected methods, and protected variables
  - we will see much more about inheritance in later lectures

javadoc

- It is a good habit to document the parameters and return values of each public method

```
public class OneString {
  private String myString;
  /** accessor method for this object
      @return the value stored in this object
   */
  public String get() {
    return myString;
  }
  /** change the value in this object
      @param newValue the value to be saved
   */
  public void set(String newValue) {
    myString = newValue;
  }
}
```

- `/* this is a comment that may extend over multiple lines */`
- The first paragraph describes what the method does
- @param parameter-name describes what each parameter is for
- @return describes what the method returns
- Try this at home!!! run javadoc on the above code, and use a browser to look at the resulting documentation

Javadoc usage

- Javadoc is strongly recommended for all your public methods
- it is standard documentation in many software development businesses and in most open-source projects
- it gives basic information to anybody using your methods

Need for Constructors

- Instance variables should be initialized, just like any other variable
- a regular initialization is fine:

  private int x = 99;
- sometimes we want the instance variables to be initialized differently for different objects
- for example, I may want a OneString where the string is initialized to "hello", and a different OneString that initializes it to "world"
- constructors allow us to specify these initial values when the object is created with `new`

Default Constructor

- if the programmer does not define a constructor, the instance variables are still initialized
  - to their type-specific default value
  - e.g. 0 for int
- this is called the default constructor, and Java only provides it if the programmer does not provide an explicit, no-arguments constructor

Programmer-Defined Constructors

- Let’s add a constructor to the OneString class:

```
public class OneString {
  private String myString;
  public OneString() {
    myString = new String("");
  }
  public OneString(String init) {
    myString = init;
  }
  public OneString(String init, int repeat) {
    myString = "";
    while (repeat-- > 1) {
      myString = myString + init;
    }
  }
```

- the constructor must have the same name as the class
- a constructor has no return type
  - you can return from a constructor, but cannot return a value
- we can have multiple constructors with different parameter types
  - Java uses the types to figure out which constructor you are calling, so you cannot have multiple constructors with the same parameter types
  - having multiple methods with the same name (but different parameter types) is called **overloading**
- it’s ok to overload methods in general, and not just constructors

differences between constructors and methods

- it is not possible to call a constructor directly
  - unlike a regular method
- a constructor is only called when using new
- constructors don’t have a return type

Object references and `null`

- Every reference points to the memory for an object
- every reference except for one, the special value `null`
- `null` just means an object reference that doesn’t refer to any object
- `null` is also the default initialization value for every object reference
- trying to use an instance method on a null object reference results in throwing a NullPointerException

```
String s;
if (s.length() == 0) {    // throws NullPointerException
```

- or, to avoid the exception, first test whether a reference is null:

```
if ((s != null) && (s.length() == ...
```

Empty and `null` strings

- `String s1 = null;`
- `String s2 = "";`
- `String s3 = new String("");`
- what is the difference between these three string values?

Empty and `null` strings part 2

- `String s1 = null;`
  - this string variable does not refer to any object
- `String s2 = "";`
  - this string variable refers to an object that is a string of length 0. It is not a newly created object
- `String s3 = new String("");`
  - this string variable refers to a newly created object that is a string of length 0
- the book suggests always initializing instance variables so they do not default to `null`

Summary

- instance variables are stored in the memory of an object, and can take different values in different objects
  - the main job of a constructor is to initialize all the instance variables
  - accessor and mutator methods provide controlled access to private variables
- instance methods are not static, and can access instance variables
  - methods (not just constructors) can have the same name as other methods, as long as the parameter types are different
    - we say that these methods are overloaded
- instance variables are always private, methods are public if possible
- `null` is a legal value for any Object reference
- use Javadoc, it is good practice