CMSC131

Objects: Designing Classes and Creating Instances

Classes and Instances of Objects

- We have seen that each Java application we run has a **`static main`** method that serves as the starting point.
  - The class that contains this method will sometimes contain other static helper methods.
- We have seen classes which contain only static methods.
  - **`FlagMaker`** from the project
- We have also seen that there are stand-alone classes from which we can instantiate objects, such as the **`String`** class.
- Today we will talk about creating our own classes for instantiation…

Declaring a new Class

The class needs to be in a .java file with the same name as the class.

file: Student.java

```
public class Student {

}
```

Static vs. Instance Variables (I)

Our class might have added to it a…

- static variable, in which case all instances of the class share the same single copy of that variable (so the data is not part of any individual object of that class type)

- instance variable, in which case each instantiated object of the class type has its own unique copy of that variable within it

Static vs. Instance Variables (II)

```
file: Student.java
public class Student {
  //Instance Variables
    //Private variables can not be directly
    // accessed by "outside world"
    private String name;  //a reference
    private String ident; //a reference
    //Public variables can be directly accessed
    public int tokenLevel;

   //Static Variables - must be initialized here
    private static int currentCount = 0;
    private static int overallCount = 0;
}
```

Instantiating and Using Class Objects

With most classes we will need to both declare a variable that will refer to an object and also instantiate an object for it.

```
Student s1 = new Student();
```

We can then access any public variables and methods using "dot" notation.

```
s1.tokenLevel = 6;
```

Static vs. Instance Methods (I)

Our class might have some

- static methods, which can be invoked either via the class name or via an object of that class type, but the method can only access variables which are static members of the class

- instance methods, which can be invoked only via an instantiated object of that class type, and it can access both variables which are static and variables which are instance

Static vs. Instance Methods (II)

file: Student.java

```
public class Student {
  public void receiveTokens(int tokenLevel) {
     this.tokenLevel += tokenLevel;
  }

  public static int howManyEver() {
     return overallCount;
  }
}
```

The first exam is…

1. Wednesday, October 10th during LECTURE time
2. Wednesday, October 10th during LECTURE time
3. Wednesday, October 10th during LECTURE time

Which kinds of variables are contained as part of the individual objects?

1. instance
2. static
3. both
4. neither

Constructors

When a new instance of an object is created, its instance variables can be initialized via a constructor.

There can be multiple constructors, each with different parameter lists (this is known as overloading a method).

Consider what constructors we might want for this **Student** class we've been building.

```
public Student(String nameIn, int tokensIn, int identIn)
public Student(String nameIn, int identIn)
public Student()
```

"Primary" Constructor for **Student**

```
public Student(String nameIn, int tokenLevel, int
   identIn) {
   //In the following three lines, we assign values to
   // the instance variables of the newly create object.
   String name = nameIn;
   this.tokenLevel = tokenLevel;
   ident = convertIntToString(identIn);

   //In the following two lines we increment the static
   // variables that all instances of the class share.
   currentCount++;
   overallCount++;
}
```

NOTE: `convertIntToString(identIn)` **is a static method written as part of our** `Student` **class.**

```
Student s2 = new Student("Evan", 27, 112684532);
```

`this`

In object oriented languages like Java and C++ it is often useful to have a way to refer to the object that invoked an instance method when inside that method.

Within an instance method, we have **`this`**.

In the previous code example, the line

```
name = nameIn;
```

could have also been written as

```
this.name = nameIn;
```

Multiple Constructors

We can make use of **`this`** to reduce redundant code in writing multiple constructors.

```
public Student(String nameIn, int identIn) {
  this(nameIn, DEFAULT_TOKENS, identIn);
}

public Student() {
  this("Unknown", DEFAULT_TOKENS, DEFAULT_IDENT);
}
```

These both invoke the "primary" constructor, filling in "missing" values that are needed to initialize the object's variables.

"Usual Suspects"

There are some instance method that are "expected" when writing Java classes, such as…

```
public String toString() {
  return
   "Name: " + name +
   ", ID: " + ident +
   ", Tokens: " + tokenLevel;
}
```

"Usual Suspects"

There is another instance method that is "expected" when writing Java classes, which for now we will express as the following but which we will revist soon.

```
public boolean equals(Student otherStudent) {
  return
   ident.equals(otherStudent.ident) &&
   (tokenLevel == otherStudent.tokenLevel) &&
   name.equals(otherStudent.name);
}
```

`public` vs. **`private`**

- We will explore the reasons why we might make some variables and methods public or private as we see more about object oriented programming.
- In the **`Student`** example, let us consider the **`ident`** value it stores.
  - It is "sent" to the constructor as an **`int`**.
  - The instance stores it as a **`String`** internally.
  - Because it is a private variable, only methods within this class can access it directly.
  - This means we could change how it is stored and as long as the methods we write for the class are rewritten to access the new storage decision, the "outside world" will never know the difference.

Copy Constructor

There are times when an existing object is used to initialize a newly created object, in which cases Java looks for a copy constructor.

```
public Student(Student aStudent) {
  this(
     aStudent.name,
     aStudent.tokenLevel,
     convertStringToInt(aStudent.ident)
  );
}
```

NOTE: Because we store the **`ident`** as a **`String`** internally but the "primary" constructor expects an **`int`**, we convert it back before sending it here…