CMSC131

Creating a Datatype Class

Continued Exploration of Memory Model

Reminders

- The name of the source code file needs to match the name of the class.
- The name of the constructor(s) need(s) to match the name of the class and have no return type.
- Instance fields appear inside each object while static fields are shared and live inside metaspace.
- Good style keeps fields private and provides getters and setters as needed.

Classes and Instances of Objects

Let's imagine we want to:

- Have some instance fields per object:
  - Full Name (String)
  - Nickname (StringBuffer)
  - UID (String with "right" format)
  - Midterm Score (float)
- Have some static fields at the class level:
  - Keep track of how many have been created.
  - Keep track of how many still around?

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 {


}
```

When does the name of the .java file in which a public class is defined have to match the class name?

- 33% A. Never
- 33% B. Usually
- 33% C. Always

Response Counter

Static vs Instance Fields

```
file: Student.java
public class Student {
  //Instance Fields
     private String name;  //a reference
     private StringBuffer nickname; //a reference
     public String uid; //###-##-####
     private float semesterScore; //a primitive

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

NOTE: The **uid** is public in this example just so we can see the impact that decision has.

Some Static Helpers

Before starting on the instance methods, we will create two helpers to deal with the way we would like to store formatted UIDs internally.

```
//See posted code for implementation details of
//  these two methods.
private static String convertIntToString(int inVal)
private static int convertUIDToInt(String inVal)
```

Setters and Getters

A common style is to provide public methods known as getters and/or setters for any fields with which you want users to be able to interact. These should be careful to make independent copies of things to avoid data corruption within the objects. In this case, the standard ones to provide are:

```
setName(String)          getName()
setNickname(String)      getNickname()
setUID(int)              getUID()
setScore(float)          getScore()
```

Code: Setters and Getters

```
public
void setName(String nameIn){
  name=nameIn;
}
```
	```
	public
	String getName() {
	  return name;
	}
	```

```
public
void setNickname(String nickIn) {
  nickname=
        new StringBuffer(nickIn);
}
```
	```
	public
	String getNickname() {
	   return nickname.toString();
	}
	```

```
public
void setUID(int uidIn){
  uid=convertIntToString(uidIn);
}
```
	```
	public
	int getUID() {
	  return convertUIDToInt(uid);
	}
	```

```
public
void setScore(float scoreIn) {
  semesterScore=scoreIn;
}//Note that our team decided on float.
```
	```
	public
	float getScore() {
	  return semesterScore;
	}
	```

`this`

Within an instance method you can optionally refer to "the object that invoked this method" using the automatically created local reference **`this`**.

```
public void setName(String nameIn){
  this.name=nameIn;
}
```

Without the use of **`this`** java looks for the "best" variable with that name in the correct scope. Why might that be a problem?

Local Shadow Variables

Within a method, if you declare a variable it will have a “higher priority” if java is looking for the best match.

That means a local variable is “seen” by default over a field of the object.

The following code works, but it is considered poor/risky style. Do you see why?

```
public void setName(String name){
   this.name=name;
}
```

Local Shadow Variables (II)

Within a method, if you declare a variable it will have a “higher priority” if java is looking for the best match.

That means a local variable is “seen” by default over a field of the object.

The following code does not work. Do you see why?

```
public void setName(String name){
   name=name;
}
```

An additional modifier

For this class, let’s say we want to provide a modifier that allows a **`Student`** object to have something appended to the existing nickname…

```
public void appendNickname(String suffix) {
   nickname.append(suffix);
}
```

Instantiating Class Objects

In order to usefully be able to create objects of this type we will probably want to provide some 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).

Typical Style Constructor

The most natural would be one that takes in all of the required information:

```
public Student(String nameIn, String nickIn, int uidIn)
```

but we will also create a few others as examples of syntax and future usage.

Code: Typical Style Constructor

```
public Student(String nameIn, String nickIn, int uidIn) {
  //Use our setters to initialize the fields.
  //  This allows us to avoid duplicate/redundant
  //  code in our class.
  setName(nameIn);
  setNickname(nickIn);
  setUID(uidIn);
  setScore(0);

  //Increate the values in the static counters
  currentCount++;
  overallCount++;
}
```

Additional Constructor Options

```
//You can provide several constructors as long as their signatures
//  are different.  Java will call the "best" one automatically when
//  a new object is requested.

public Student(String nameIn, int identIn) {
  //Another constructor of the class can be invoked
  //  using the keyword this as a method name.
  this(nameIn, DEFAULT_NICKNAME, identIn);
}

public Student(String nameIn) {
  this(nameIn, DEFAULT_NICKNAME, DEFAULT_UID);
}

public Student() {
  this("Anonymous"+currentCount,DEFAULT_NICKNAME,DEFAULT_UID);
}
```

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 existingStudent) {
  this(
    existingStudent.getName(),
    existingStudent.getNickname(),
    existingStudent.getUID()
  );
  //What's missing?
}
```

NOTE: We use getters to avoid having duplicate/redundant code and because we were careful to have our getters ***not*** create aliasing risks.

Shallow copies in a constructor

What would the following version of a copy constructor do in terms of memory?

```
name = existingStudent.name;
nickname = existingStudent.nickname;
UID = existingStudent.UID;
semesterScore = existingStudent.semesterScore;
```

Reference/primitive copies done of each field is called a shallow copy. While this is fine for primitive values and immutable objects, it can be risky with mutable objects (like the nickname) is we had wanted a fully independent copy made!

Instantiating 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 exampleStudent =
   new Student("Fred", "Freddie", 123456789);
```

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

```
  exampleStudent.uid = "Not a valid UID.";
```

This is why it is suggested to make fields non-public; you “control” them.

Which of the following are contained as part of the individual objects?

(click all that apply)

- 🟩 17% 1. Reference to the name.
- 🟩 17% 2. Reference to the nickname.
- 🟥 17% 3. The UID as an integer.
- 🟩 17% 4. Reference to the UID.
- 🟥 17% 5. The currentCount integer.
- 🟩 17% 6. The score as a float.

Response Counter

Draw what this example looks like…

What happens to the **stack**, the **heap**, and **metaspace** when the following main method executes up until the …?

```
public static void main(String[] args) {
  Student exampleStudent =
    new Student("Fred", "Freddie", 123456789);
  …
}
```

Static Getters

```
file: Student.java
public class Student {

  public static int howManyNow() {
      return currentCount;
  }

  public static int howManyEver() {
      return overallCount;
  }

}
```

Standard `toString()` Method

Let’s look at two different ways to implement the string construction in the body (blue vs green).

```
@Override public String toString() {
  return
    "Name: " + name + "(" + nickname + ")" +
       ", ID: " + uid +
       " [" + semesterScore + "]";
    "Name: " + getName() + "(" + getNickname() + ")" +
       ", ID: " + uid +
       " [" + getScore() + "]";
}
```

Which style do you prefer?

0% A. Using the fields

0% B. Using the getters

```
"Name: " + name + "(" + nickname + ")" +
    ", ID: " + uid +
    " [" + semesterScore + "]";
"Name: " + getName() + "(" + getNickname() + ")" +
    ", ID: " + uid +
    " [" + getScore() + "]";
```

Response Counter

Java-expected **equals** Method

```
@Override
public boolean equals(Object otherObject ) {
  //The next three lines are things we will explore later.
  if (otherObject == null) {return false;}
  if (otherObject.getClass() != this.getClass()) {return false;}
  Student otherStudent = (Student)otherObject;

  //The code of interest to us right now.
  return (
    name.equals(otherStudent.name) &&
    nickname.equals(otherStudent.nickname) &&
    uid.equals(otherStudent.uid) &&
    semesterScore==otherStudent.semesterScore
  );
}
```

`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 **`uid`** value it stores.

- It is "sent" to the constructor as an **`int`**.
- The instance stores it as a **`String`** internally.
- If it is a private field then only methods within this class can access it directly.
- This means others can’t mess with the format and 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.

Testing Your Class

When building a new class, you should test all functionality it provides.

The posted **`StudentTesting`** class has some examples that you can open in Eclipse and explore…

`Student` objects in Code

To explore, you can go into Eclipse and look at a few code examples where Student objects are created and used…

```
Student s1 = new Student();
Student s2 = new Student("Sam", 123456789);
Student s3 = new Student("Pat", "Patty", 987654321);
Student s4 = new Student(s3);
Student s5 = s3;
  //What's going on here in memory?
s3.appendNickname("IsCool");
s3.setScore(s3.getScore()+10);
  //What's going on here in memory?
s3 = new Student("New Student!!!", 345363267);
  //What's going on here in memory?
```

Copying **`Student`** objects

One place where we needed to think about whether an object is mutable is when we talk about aliasing and making copies of things.

Again, consider our copy constructor and what could be happening in memory.

Our objects have:

```
String name
StringBuffer nickname
String UID
float semesterScore
```

Shallow Copies and Mutable Objects

Image a class that contains references to two student objects:

```
public class StudentPair {
   Student stu1;
   Student stu2;
   …
}
```

Now imagine we had the following copy constructor:

```
public StudentPair(StudentPair other) {
   this.stu1 = other.stu1;
   this.stu2 = other.stu2;
}
```

If we then had code such as:

```
StudentPair sp2 = new StudentPair(sp1);
```

Where the new **`StudentPair`** created this way would contain aliases to the same **`Student`** objects as the original one. If **`sp1.stu1`** had a grade change, **`sp2.stu1`** would as well!

Deep Copies and Mutable Objects

Image a class that contains references to two student objects:

```
public class StudentPair {
   Student stu1;
   Student stu2;
   …
}
```

Now imagine we had the following copy constructor:

```
public StudentPair(StudentPair other) {
   this.stu1 = new Student(other.stu1);
   this.stu2 = new Student(other.stu2);
}
```

We could then have code such as:

```
StudentPair sp2 = new StudentPair(sp1);
```

the new **`StudentPair`** created this way would contain a reference to different **`Student`** objects as the original one. If **`sp1.stu1`** had a grade change, **`sp2.stu1`** would *NOT*!

When objects go away…

Unlike in many other languages, java does not provide a way to explicitly say you are done with an object. However, once Java detects that there is no longer any way to access an object, it can be deleted from memory during garbage collection. When this happens, Java is supposed to execute the object’s finalize method.

```
protected void finalize() throws Throwable
{
  currentCount--;
  super.finalize(); //Java-required, covered in 132.
}
```