CMSC131

Designing Classes

"Library Classes"

Due to Java being 100% object-oriented, all code must live inside a class but there is some functionality and/or information that might be best kept in a more central location.

- Consider mathematical concepts such as Floor that can be applied to any numeric value and constants such as π.

Classes containing only static fields and methods (eg: **`java.lang.Math`**) are sometimes called "library classes" and no objects can be created of that type.

"Datatype Classes"

Most classes that you will encounter and build will be "Datatype Classes" where the class defines a structure to hold information as well as functionality on objects with that structure.

We have also seen that there are generally useful classes from which we can instantiate objects, such as the **`String`** class.

Today we will explore the general thought process and syntax of creating our own datatype classes for instantiation…

Designing a Datatype Class (I)

Some general considerations when designing a new datatype class:

- Which fields should be **static** (one shared by all objects and the "outside world") versus **instance** (one per object, **"inside" the object**).
- Which methods should be **static** (not associated with a particular object, only have access to static fields) versus **instance** (have to be invoked by an object, have access to instance fields of that object as well as all static fields).

Designing a Datatype Class (II)

Additional considerations when designing a new datatype class:

- How might objects need to be initialized by our code when first created by a "new" instruction.
- What "standard" functionality should be built.

Static Fields

With static fields all instances of the class, as well as any other part of the program, share the same single copy of that variable (the data is not part of any individual object).

- If there is a constant value associated with the class, a **`final static`** field would be a very efficient way to provide it.
- If you wanted to keep track of how many objects of this type were create, a **`static`** field can be very useful.

Static Fields and Memory

Recall: These live in a part of Java's memory space called **metaspace**.

Once they have been allocated there, they are never deleted during the execution of the program.

Instance Fields

With instance fields each instantiated object of the class type has its own unique copy of that variable within it.

- These are more commonly used since they are what we use to hold information that is specific to an object.

Instance Fields and Memory

Recall: These live in a part of Java's memory space called **heap**.

A set of these fields are allocated each time an object of the type is instantiated and are said to live inside the object. They can be primitives or be references to other objects.

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

0% 1. instance
0% 2. static
0% 3. both
0% 4. neither

Response Counter

Fastest Responders

Seconds	Participant	Seconds	Participant

Static Methods

Static methods can be invoked either via the class name or (no recommended) via an object of that class type and the method can only access variables which are static members of the class regardless of which invocation technique is used.

Instance Methods

Instance methods 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

- These are where most functionality associated with an object is provided.

Instance Methods and their object

When working inside an instance method, you can refer to fields within the object that invoked the method using a special reference called `this` which is automatically created.

`public` vs. `private`

We will explore the reasons why we might make some variables and methods **public** or **private** (or something else) as we see more about object oriented programming.

A common approach/style is to:

- Make all fields non-public and provide public getter/setter methods if other parts of the program can access them.
- Only make officially supported methods that you want other parts of the program to directly access be **public** ones.

Constructors

When a new instance of an object is created, its instance variables can be automatically initialized via a special type of method called a constructor.

- The name of the constructor method is the same as the name of the class. There is no return value (not even **void**).
- There can be multiple constructors, each with different parameter lists (this is known as overloading the constructor method).
- A highly recommended one is called a copy constructor and is used to make a duplicate of an existing object.

Common Standard Functionality

There are two instance methods that are considered "standard" to provide when writing Java datatype classes:

```
public String toString();
```

This method is what Java will use (for example) if you try to print the object.

```
public boolean equals(Object other);
```

This method is used in placed to allow program and JUnit tests to check whether the information in two objects is the same.

What if we wanted a class **Point3D**

Take a moment and think about what you think a Point3D class should have in terms of:

- data fields stored in each object
- public methods that could be invoked

Chat with your neighbor about this before we go to Eclipse to see what the full class might look like.