Basics of OOP in Java

The fundamental ideas of OOP in Java are very similar to C++: terms like class, object, method, member, static all mean what you expect them to mean. So today will be a *little* bit of review, but also some possibly-new ideas on what OOP "really" means—it's all about design.

Basic Vocabulary

In Java, all code lives in a class. (Compiled files are called class files!) And with the exception of the main() method (the "driver"), everything is organized around objects—in essence, a Java program is a collection of objects which each know how to do some stuff, and who know how to send messages to other objects to do other stuff.

Basic vocabulary:
Object: collection of data and functions which manipulate that data. The data is often referred to as "**instance variables**" or "**members**;" the functions are almost always called "**methods**."

Class: essentially, a "template" for a particular type of object. Declares and defines members and methods, as well as special methods called **constructors**, which describe how to create new objects from a set of data. Classes may also contain extra code for keeping track of the objects of that class: **static members** and **static methods** are code "belonging to the class" rather than to any particular object.

One nice thing about objects is that they can conceal their "inner workings" from outsiders—maybe there's data that only the object knows how to manipulate, or methods only the object should be able to call. These are declared as **private** — only objects of the same class can use these methods. There are three other levels of visibility: if we don't say anything about visibility, then any code in the same **package** can access the method/member, but objects outside the package cannot. **Public** elements can be accessed by anything. And **protected** we'll talk about shortly.

Constructors have a special syntax; their name is the same as the class. Every class automatically has a **default constructor**, which is the constructor with no arguments. You don't have to write this constructor; the compiler will provide it for you, but if you want to be sure every object has a certain property, you may want to provide your own implementation.

You can add as many constructors that take arguments as you like (as long as the compiler can discern which one you mean). Look at my `Card` constructors. You can also invoke other constructors within constructors—this is a *very* good idea, because it helps keep behavior consistent by minimizing the amount of duplicate code you have to write. The only catch is that you must invoke the constructor, using `this()`, before you do anything else. (See p 152)

(And, similarly, you can declare multiple 'versions' of one method, using the same name but different parameters. This lets you, for example, offer multiple ways of doing things for whoever is using your code. This concept, of multiple versions of the same thing, distinguished by their signatures, is called **overloading**.)

Toward Object-Oriented Design

So far, this isn't that different what you might have learned in a introductory course that didn't really focus on OOP—code is organized into classes/methods instead of (say) modules and functions, but the basic way of thinking isn't too different. Just you wait.

The challenging (and fun) part of OOP is developing a *design* in which classes and objects have the "right" set of relationships among themselves.

The most intuitive relationship is the "HAS-A" relationship, also called "**composition**." For example, the Deck HAS-A Card (52, actually). So an object can be "contained inside" another object.

The other major relationship is the "IS-A" relationship; for example, a Cheeseburger IS-A Hamburger-- which is to say, it's a special hamburger, with all the hamburg-y goodness, plus extra components and behaviors. This is generally a trickier relationship to work with: but vocabulary wise, we can say extends, **subclasses**, **is derived from**,…. This relationship is often drawn as a kind of tree relationship—see the tree of Exceptions on p 106.

Declaring that a class extends another means that objects of the subclass have everything the 'root'/base class has, plus they can add or change: objects in the subclass "**inherits** variables and methods from the superclass". They can simply declare new members/methods, too. But the real power comes when you **override** methods of the superclass: if the subclass declares a method with exactly the same name and arguments as in the base class, then you are essentially specifying that the specialized objects of the subclass behave differently from the objects of the superclass. And this is VERY cool. (see p 166).

For example:

```
class Hamburger {

        int getCalories();
}

class Cheeseburger extends Hamburger {

        int cheeseSlices;

         // this needs to depend on the amount of cheese
        int getCalories() {
                return super.getCalories() + 100 * cheeseSlices;
        }
}
```

And then this is fantastic:

```
Hamburger[] breakfast = new Hamburger[4];

breakfast[0] = new Hamburger();
breakfast[1] = new Hamburger();
breakfast[2] = new Cheeseburger();
```

```
breakfast[3] = new Cheeseburger();
```

First: can I put a **Cheeseburger** in an array of **Hamburgers**? Yes! A **Cheeseburger** IS-A **Hamburger**.

```
System.out.println("Your total calories for the meal: ");
int calories=0;
for (Hamburger h : breakfast) {
        calories += h.getCalories();
}
System.out.println(calories);
```

But then how do I make sure the right **getCalories()** is called? The JVM always uses the *most specific* version of any overridden method, no matter what type the object seems to be.

This is sometimes called *subtype polymorphism* — objects will act like different types in different contexts.

There's another way to achieve polymorphism in Java. Sometimes you want a class to be "derived from" several others—often, you want object to offer several kinds of behaviors, and you want to be able to see the object through different lenses. For example, you might want to have a **Hamburger** which is also a **Liquid**. Such a class could/should inherit from both **Hamburger** and **Liquid** classes. In C++, you can do "multiple inheritance," but in Java, you can only **extend** from one class. Instead, Java lets you declare an **interface** which a class can **implement**. So, for example, I could write[1]

```
class MeatSmoothie extends Hamburger implements Liquid {

}
```

Now, what is an interface? It's simply a list of methods that an implementing class must implement, such as:

```
interface Liquid {
        void freeze();
        void drink() throws TooFrozenToDrinkException;
}
```

thus, **MeatSmoothie** inherits **getCalories()** from **Hamburger** and also has to define **freeze()** and **drink()**.

```
Class MeatSmoothie extends Hamburger implements Liquid {
        void freeze() {...}
```

---

1 Note: If you don't believe that a hamburger thrown into a blender and pulverized is still actually a hamburger, then it wouldn't make sense to make **MeatSmoothie** a subclass of **Hamburger**. Instead, it would be better to give **MeatSmoothie** a **Hamburger** member—that is, to use **composition** instead of **derivation**.

```
        void drink() throws TooFrozenToDrinkException {...}
}
```

And then we can have

```
Liquid[] beverages = new Liquid[2];

beverages[0] = new Soda();
beverages[1] = new MeatSmoothie();

for (Liquid l : beverages) {
        l.drink();
}
```

In Java, interfaces are usually used to describe groups of behaviors that correspond to a property. For example

```
interface Iterable {
        Iterator        getIterator(); // slightly inaccurate
}

interface java.awt.Shape {
        boolean             contains(double x, double y)
        boolean             contains(double x, double y, double w, double h)
        boolean             contains(Point2D p)
        boolean             contains(Rectangle2D r)
        Rectangle           getBounds()
        Rectangle2D         getBounds2D()
        PathIterator        getPathIterator(AffineTransform at)
        PathIterator        getPathIterator(AffineTransform at, double flatness)
        boolean             intersects(double x, double y, double w, double h)
        boolean             intersects(Rectangle2D r)
}
```

But the IS-A relationship can be slippery: what if you want to create a class which IS-A subclass of something that doesn't really exist?

Let's look at the `WarAndPeace` lab. It says that `PeaceCard` should extend `Card`. But is a `PeaceCard` actually a *more specialized version* of a `Card`? No; it's just something acts a bit differently; it's not more specialized. It might make more sense rename `Card` to `WarCard`, then to say that both `PeaceCard` and `WarCard` extend a superclass called `Card`.

Let's start working on this...

```
abstract class Card {
```

```
	private CardValue value;
	private CardSuit suit;

	public Card (CardValue cv, CardSuit cs) {
		this.value = cv;
		this.suit = cs;
	}

	public Card (CardSuit cs, CardValue cv) {
		this(cv,cs);
	}

	public String toString() {
		return value + " of " + suit;
	}

	abstract public boolean winner(Card c);
}

class WarCard extends Card {

	public boolean winner (Card c) { ...}

}

class PeaceCard extends Card {

	public boolean winner (Card c) { ...}

}
```

What do we do about **winner**? We don't know; it doesn't make any sense to pick an implementation —if we do, there's a risk that the subclasses won't bother to override it. What we can do instead is admit that it doesn't make any sense to define **winner()** at this level. Instead, we'll declare it as **abstract**, and leave off the implementation. Because this means Card isn't a “fully developed” class, we must also declare it as abstract. Then the compiler will make sure that when we write **PeaceCard** and **WarCard**, we provide implementations for **winner()**.