Inheritance

Overview

Inheritance describes the relationship between Java classes. Java classes can inherit (extend) other classes. You can think of it as the class (child class) is an extension of the other class (parent class). The relationship is similar to concentric circles, where the child class *inherits* all the properties of the parent class along with whatever properties it has itself, just like an outer circle encompasses the inner circles.

```
public class Animal {
    public void walk() {
        ...
    }
}
public class Dog extends Animal {
    public void bark() {
        ....
    }
}
```

In this example the Dog class extends the Animal class, so it has all the properties of the Animal class along with the properties of the Dog class. This means that a Dog object can access the walk() method in the Animal class. You can think of this as an "is" relationship. A dog "is" an animal, so it should be able to do all things animals can do. Generally you can always think of it as "child is parent".

Overridden & Overloaded Methods

```
public class Dog extends Animal {
    public void walk() {
        ...
    }
    public void bark() {
        ....
    }
    public void bark(String s) {
        ...
    }
}
```

If the Dog class looked like this, then it would contain both overridden and overloaded methods. It's important to understand method signatures before continuing.

METHOD SIGNATURE PICTURE

Overridden Method: A method that has the exact same method signature as another method in the parent class or up the hierarchy.
Overloaded Method: A method that has the same name as another method in the **same** class, but accepts different parameters.

Java can tell which overloaded method to use just by looking at what parameters are passed in. Overridden methods require knowledge about static and dynamic types to fully understand how Java executes code with them.

Static and Dynamic Types

Java objects have two types, static and dynamic. The static type is the type of the variable pointing to the object (left of =) and the dynamic type is what type the object actually is (right of =). When initializing

objects in Java, the static type must be the same as or higher up the class hierarchy than the dynamic type.

STATIC DYNAMIC TYPE PICTURE

Example
You can think of Java objects as boxes with a label and the actual object inside. The label is the static type and the actual object is the dynamic type. This label-object relationship needs to be correct or Java will complain. Going back to Animals and Dogs, this initialization would be acceptable

```
Animal a = new Dog();
```

but this would not be

```
Dog d = new Animal();
```

This is because the animal is not necessarily a dog, while a dog is precisely an animal. Going back to the "is" relationship, you can think of it as "dynamic type is static type".

Dynamic Method Selection (DMS)

Java uses dynamic method selection to select the appropriate method to execute. It's not very complicated, but can be tricky if not fully understood. When you run your code, Java goes through two processes, compile time then run time.

Compile Time: The compilation of the Java code.
Run Time: The actually running of the Java code.

During compile time, Java looks at the **static** type of the object calling the function. It then searches that class for the function **with the same name** as the function that the object is trying to call. If not found in that specific class, Java will go up the class hierarchy until it finds it or it cannot, in which a compile time error will be thrown. **Java will then remember this method signature.**

During run time, Java will start in the class of the **dynamic** type and look for the **that same** method signature. If it finds it in the dynamic type class, then it'll execute that version of the method. If it cannot find a function with that same signature, it will go up the class hierarchy until it finds a function with the same signature and then executes that version of the function.

***Note: if Java is trying to execute bark(Dog d) and finds bark(Animal a) during compile time, that method signature will be the one saved since Dogs are Animals**

Casting

Casting is a way to "change" the static type of an object for execution purposes. It is done by adding "(class)" in front of the object, where "class" is the class we are casting the object to.
Example:

```
Animal a = new Dog();
((Dog) a).bark();
a.bark();
```

If we tried running **a.bark();** without the cast, Java will throw a compile time error because it'll look for **bark()** in the Animal class and won't find it even though we know the object is dynamically a Dog and can bark. Casting solves this issue because we are essentially telling Java that "object a is a Dog, trust me". Java will then look in the Dog class during compile time, find the method, and execute it appropriately. Going back to the box and label analogy, think of casting as relabeling the box temporarily.

***Note: function and method are two words that describe the same thing**

TL;DR

1. Child "is" (inherits from) parent
2. Overridden: same **signature** as another method up the hierarchy
3. Overloaded: same method **name** as another method in the same class
4. Static type is at least (as general as) dynamic type
5. Static type is the type of the variable reference and the dynamic type is the instance of the object (static var = dynamic obj for assignment statements)
6. Static type is what is claimed to be in the box (label) and dynamic type is what is actually in the box
7. DMS:
   (a) Look, in static type class, for a method with the same name as that of the desired function
   (b) If found, remember that signature. If not, go up class hierarchy until found, else compile time error
   (c) Look, in dynamic type class, for that **same** signature
   (d) If found, execute that function. If not, go up class hierarchy until found, else run time error
8. Casting temporarily changes the static type of a reference (sticks a new label on the box)