Inheritance and Generics

1 Inheritance

Inheritance is the idea that a specific object can have all the behaviors and properties of its parent. In order for something to inherit from a parent, you must use the *extends* keyword. For example:

```
1 class Superhero{...}
2 class Batman extends Superhero{...}
```

In the above example, the Batman class will have all the attributes of a Superhero. Additionally Batman will be able to have its own attributes. This leads us to a key point. The is-a relationship. The class that extends a parents class "is-a" the parent class (in this case Batman is a Superhero). However, a Superhero is not Batman because the Superhero class does not extend Batman.
We know that when a class inherits from another, it inherits the methods, but a subtle point is that when you instantiate the child class, an implicit call to *super()* is made in the child constructor as the first call. This means that if the parent class has a constructor that takes in no parameters, its contents will get run before the the child's constructor. If you wish to run a parent constructor that has arguments, simply write *super(your arguments here).* To call any other parents methods from a child's method use *super.parents method name(parent's method arguments);*
We have gone over class Inheritance, now we will discuss interface inheritance. For interfaces, we use the word *implements* instead of extends.

```
1 interface Plumber{...}
2 class Mario implements Plumber{...}
```

Similar to a class, the interface creates an is-a relationship. That a class that implements the interface is-an instance of the interface.
Though on the surface, it may seem that extending classes and implementing interfaces do the same thing, they serve very different purposes, and it is important to see the distinctions. One of the biggest differences is that when a class extends another class, it does not have to re implement any methods. Assuming you did not have the same method signature in your subclass, calling a method will result in the parent's method being called without a compilation error. Interfaces on the other hand serve more as a blueprint for code that needs to be implemented. The only exception to this rule is if a method in the interface is declared with the *default* keyword and has a body. Another distinction between the two philosophies is that a class can only *extend* from one class; however, it can *implement* multiple interfaces.
Now that we have gone over inheritance, we can bring back static and dynamic types. When we are declaring a variable, the class on the left is the *Static Type* and the class on the right is the *Dynamic type.* When declaring a variable, you must make sure that the item to the right is-a version of the item on the left of the equal sign. ***This is a reference class for the next few examples***

```
1  public class Fellow{
2      String name;
3      public Fellow(){
4             this.name = "I'm nameless"; }
5       public void breathe(){
6              System.out.println("Breathing noise");
7       }
8  }
9  public class Professor extends Fellow{
10     public Professor(){}
11     public void teach(){ System.out.println("I taught"); }
12 }
```

Now let's walk through a few function calls:

```
1 Fellow josh = new Professor();
2 System.out.println(josh.name);
3 josh.breathe();
4 josh.teach();
```

1st line creates a Professor whose static type is a Human, 2nd line prints "I'm nameless", 3rd line prints "Breathing noise", 4th line is a compilation error.

In the above example, *josh.teach()* gave us a compilation error because even though josh's dynamic type is a teacher, it's static type is a human. When making a method call, Java checks to see if the method exists in the static type, if it doesn't it raises a compilation error. To get around this, we use a trick called casting. Basically, when we know something the computer doesn't we tell the computer we know what the object really is. to make the above code work we would write

```
1 (Professor josh).teach();
```

If however, the method does not exist in the casted type, we will get a runtime error. So be careful with casting!

2 Generics

Let's say that we want to have some unknown object be a variable inside of a class. Inside of the class header, we can declare unknown types that we know will be variables. Take the following class as an example.

```
1 public class DictionaryItem<K, V> {
2     K key;
3     V value;
4
5     public DictionaryItem(K key, V value) {
6         this.key = key;
7         this.value = value;
8     }
9 }
```

It is not totally obvious why this would be useful, why use this when we can just have a normal class that looks cleaner? The reason why can be seen below.

```
1 DictionaryItem<String, String> webster =
2 new DictionaryItem<String, String>("Kartik", "Kapur");
3         DictionaryItem<Character, Integer> numberDict =
4         new DictionaryItem<Character, Integer>('a', 1);
```

If we want to have different types of a Dictionary, we may not want to rewrite the whole class. This is why generics are so useful.
Similarly, if we want to make just one method have a generic type, we can do that with a similar declaration.

```
1 public static<Key, Value> Dictionary <Key, Value> add(Key k, Value v){...}
```

The initial <Key, Value>refer to the Dictionary return value and tell us what types will be in the Dictionary. The second <Key, Value>refers to the types that are the being passed in for the arguments into the method. Generics can be tricky, so it is important to keep track of when and how you declare.