Interfaces

- An interface is a collection of constants and abstract methods
- The methods in an interface are always public and abstract
- The constants in an interface are always public static and final
- You don't need to specify these attributes for the methods and the constants in the interface
- To make use of an interface, you implement the interface in a class
- You declare that the class implements the interface and you write code for each of the methods declared in the interface as part of the class definition
- Any constants in an interface are available to the class implementing that interface

Example: Interface

```
public interface Conversions {
  double inchesToMillimeters (double inches);
  double ouncesToGrams(double ounces);
  double poundsToGrams(double pounds);
  double hpToWatts(double hp);
  double wattsToHP(double watts);
}
```

Example: Interface Implementation

```
import static conversions.ConversionFactors.*;   // Import static
members

public class TryConversions implements Conversions {
  public double wattsToHP (double watts) {
    return watts*WATT_TO_HP;
  }
  public double hpToWatts (double hp) {
    return hp*HP_TO_WATT;
  }

  public double ouncesToGrams(double ounces) {
    return ounces*OUNCE_TO_GRAM;
  }

  public double poundsToGrams(double pounds) {
    return pounds*POUND_TO_GRAM;
  }

  public double inchesToMillimeters(double inches) {
    return inches*INCH_TO_MM;
  }
```

Example: Interface Implementation

```
public static void main(String args[]) {
    int myWeightInPounds = 180;
    int myHeightInInches = 75;

    TryConversions converter = new TryConversions();
    System.out.println("My weight in pounds: "
+myWeightInPounds +
     " \t-in grams: "+
(int)converter.poundsToGrams(myWeightInPounds));
    System.out.println("My height in inches: " +
myHeightInInches
                       + " \t-in millimeters: "
                       +
(int)converter.inchesToMillimeters(myHeightInInches));
  }
}
```

Interfaces

- **Every method declared in the interface should have a definition within the class if you are going to create objects of the class**
- **Since methods in an interface are public by default you must use the public keyword when you define them in your class, otherwise your code wont compile. Why?**
- **You can implement as many number of interfaces as you like in your class**
  - public class MyClass implements MyInterface,OneIntface

Extending Interfaces

- **You can also extend an interface**
- **You can define one interface based on another by using the keyword extends to identify the base interface name**
- **Base interface is also called super interface**
- **An interface can use the contents of several other interfaces**
- **To specify an interface that includes the members of several other interfaces, specify the names of the interfaces, separated by commas following the keyword extends**
  - public interface MyInterface extends OneIntface, TwoInface
- **An interface can extend one or more interfaces but can not extend a class. Similarly an interface can not implement anything.**

Example: Extending an Interface

```
public interface DoIt {
void doSomething(int i, double x);
int doSomethingElse(String s);
}
```

```
public interface DoIt {
 void doSomething(int i, double x);
int doSomethingElse(String s);
boolean didItWork(int i, double x, String s);
}
```

What is the problem?

```
public interface DoItPlus extends DoIt {
boolean didItWork(int i, double x, String s);
}
```

Using Interfaces

- **Interfaces vs Inheritance?**

Using Interfaces

- **Where several classes share a common set of methods with given signatures but with possible different implementation, you can separate the set of methods common to the class into an interface. This interface can then be implemented by each of the class**
- **For polymorphism to work, you need one variable that is capable of referring to one of the different kind of object, as we don't have a base class now**
- **We can use a variable of the interface type as we are using the interface to specify the methods that are common to all the classes**
- **Thus if you declare a variable to be an interface type, you can use it to reference an object of any class that is declared to implement the interface(barring abstract classes)**

Example: Interface as Type

```
public interface Relatable {

// this (object calling isLargerThan) and
// other must be instances of the same class
 // returns 1, 0, -1 if this is greater
// than, equal to, or less than other

public int isLargerThan(Relatable other);
}
```

Example: Interface as Type

```
public Object findLargest(Object object1, Object object2) {
Relatable obj1 = (Relatable)object1;
Relatable obj2 = (Relatable)object2;
if ( (obj1).isLargerThan(obj2) > 0)
        return object1;
else
        return object2;
}
```

What's the difference between an interface and an abstract class in Java?

It's best to start answering this question with a brief definition of abstract classes and interfaces and then explore the differences between the two.

A class must be declared ***abstract*** when it has one or more abstract *methods*. A method is declared abstract when it has a method heading, but no body – which means that an abstract method has no implementation code inside curly braces like normal methods do.

When to use abstract methods in Java?

Why you would want to declare a method as abstract is best illustrated by an example. Take a look at the code below:

```
/* the Figure class must be declared as abstract
   because it contains an abstract method  */

public abstract class Figure
{
        /* because this is an abstract method the
           body will be blank  */
        public abstract float getArea();
}

public class Circle extends Figure
{
        private float radius;

        public float getArea()
        {
                return (3.14 * (radius * 2));
        }
}

public class Rectangle extends Figure
{
        private float length, width;

        public float getArea(Figure other)
        {
                return length * width;
        }
}
```

In the Figure class above, we have an abstract method called getArea(), and because the Figure class contains an abstract method the entire Figure class itself must be declared abstract. The

Figure base class has two classes which derive from it – called Circle and Rectangle. Both the Circle and Rectangle classes provide definitions for the getArea method, as you can see in the code above.

But the real question is why did we declare the getArea method to be abstract in the Figure class? Well, what does the getArea method do? It returns the area of a specific shape. But, because the Figure class isn't a *specific* shape (like a Circle or a Rectangle), there's really no definition we can give the getArea method inside the Figure class. That's why we declare the method and the Figure class to be abstract. Any classes that derive from the Figure class basically has 2 options: 1. The derived class must provide a definition for the getArea method OR 2. The derived class must be declared abstract itself.

A non abstract class is called a concrete class

You should also know that any non abstract class is called a ***concrete*** class. Knowing your terminology defintely pays off in an interview.

Now that we've explored the abstract method/class concepts, let's get into the concept of interfaces and how they differ from abstract classes.

Java interface versus abstract class

An interface differs from an abstract class because an interface is ***not*** a class. An interface is essentially a **type** that can be satisfied by any class that implements the interface.

Any class that implements an interface must satisfy 2 conditions:

- It must have the phrase "implements *Interface_Name*" at the beginning of the class definiton.
- It must implement all of the method headings listed in the interface definition.

This is what an interface called "Dog" would look like:

```
public interface Dog
{
        public boolean Barks();

        public boolean isGoldenRetriever();
}
```

Now, if a class were to implement this interface, this is what it would look like:

```
public class SomeClass implements Dog
{
        public boolean Barks{
        // method definition here
```

```
        }

        public boolean isGoldenRetriever{
        // method definition here
        }
}
```

Now that we know the basics of interfaces and abstract classes, let's get to the heart of the question and explore the differences between the two. Here are the three major differences:

Abstract classes and inheritance

1. Abstract classes are meant to be *inherited* from, and when one class inherits from another it means that there is a strong relationship between the 2 classes. For instance, if we have an abstract base class called "Canine", any deriving class ***should*** be an animal that belongs to the Canine family (like a Dog or a Wolf). The reason we use the word "should" is because it is up to the Java developer to ensure that relationship is maintained.

With an interface on the other hand, the relationship between the interface itself and the class implementing the interface is not necessarily strong. For example, if we have a class called "House", that class could also implement an interface called "AirConditioning". Having air conditioning not really an essential part of a House (although some may argue that point), and the relationship is not as strong as, say, the relationship between a "TownHouse" class and the "House" class or the relationship between an "Apartment" class that derives from a "House" class.

Because a TownHouse is a type of House, that relationship is very strong, and would be more appropriately defined through inheritance instead of interfaces.

So, we can summarize this first point by saying that an abstract class would be more appropriate when there is a strong relationship between the abstract class and the classes that will derive from it. Again, this is because an abstract class is very closely linked to inheritance, which implies a strong relationship. But, with interfaces there need not be a strong relationship between the interface and the classes that implement the interface.

Interfaces are a good substitute for multiple inheritance

2. Java does not allow multiple inheritance – see the discussion onJava Multiple Inheritance if you need a refresher on this. In Java, a class can only derive from one class, whether it's abstract or not. However, a class can implement multiple interfaces – which could be considered as an alternative to for multiple inheritance. So, one major difference is that a Java class can inherit from only one abstract class, but can implement multiple interfaces.

Abstract classes can have some implementation code

3. An abstract class may provide some methods with definitions – so an abstract class can have non-abstract methods with actual implementation details. An abstract class can also have constructors and instance variables as well. An interface, however, can not provide any method

definitions – it can only provide method headings. Any class that implements the interface is responsible for providing the method definition/implementation.

When to use abstract class and interface in Java

Here are some guidelines on when to use an abstract class and when to use interfaces in Java:

- An abstract class is good if you think you will plan on using inheritance since it provides a common base class implementation to derived classes.
- An abstract class is also good if you want to be able to declare non-public members. In an interface, all methods must be public.
- If you think you will need to add methods in the future, then an abstract class is a better choice. Because if you add new method headings to an interface, then all of the classes that already implement that interface will have to be changed to implement the new methods. That can be quite a hassle.
- Interfaces are a good choice when you think that the API will not change for a while.
- Interfaces are also good when you want to have something similar to multiple inheritance, since you can implement multiple interfaces.