Abstract Class:

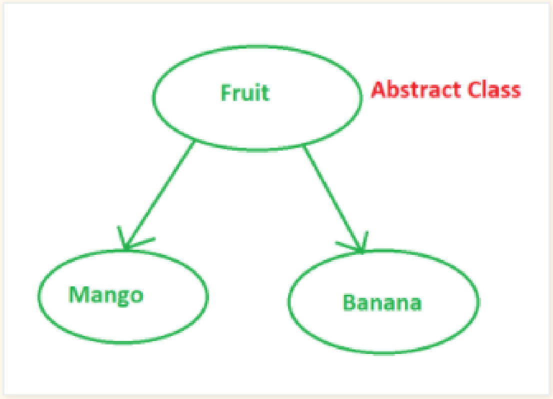
An interface is a pure abstract class, that is all. They are one in the same. I use abstract class all the time for a function in the base class that cant be implemented because it needs data that only the subclasses have, but I want to ensure that every subclass has this function and implements it accordingly.

Abstract classes let you add base behavior so programmers don't have to code everything, while still forcing them to follow your design.

Why Abstract class is Important in Java

Though there are lot of difference between abstract class and interface, key thing to remember is that they both provides abstraction. Let's take an example, you need to design a program, which can produce reports for employees e.g. how many hours they worked and how much they are paid every month. Let's assume that currently your organization only has permanent employees, which are paid monthly. You know that and you write code based upon that, after sometime your company started recruiting contract employees, which are paid at hourly rate rather than monthly salary. Now, if you need to rewrite your program to support this, they your program is not flexible enough. On the other hand, if you just needs to write some code to plug this new type of employee into system, they your program is very much flexible and maintainable. If you would have known about abstract class, you would have made `Employee` an abstract class and methods like `salary()` abstract, because that is what varies between different types of employees. Now introducing a new type of Employee would be cake walk, all you need to do is to create another subclass of Employee to represent `ContractEmployee`, and their salary method return salary based upon number of hours they had worked multiplied by their hourly rate. So, you get the idea right, we code at certain level of abstraction, which allows us to accommodate new changes in our system.

Abstract class and Method Example in Java



Let's see another *example of abstract class and method*, this one is rather tasty examples. I love fruits, they are tasty, provides vitamins and you don't need to cook them to eat. All you do, you take a fruit, wash them then you either cut them, or peel them before eating. Our example is based on this behaviour. We have created an abstract class `Fruit`, which contains some concrete behaviour like colour, whether its a seasonal fruit or not, and abstract behaviour called `prepare()`. In order to server fruits, we need to prepare them for eating, which involves either cutting them in slices or peel them. We have two concrete implementation of `Fruit` class, `Mango` and `Banana`, I chose these two because, first they are my favourites and second they have different methods of preparing them. You cut mangoes but you peel bananas.

```java
public class AbstractClassDemo{

    public static void main(String args[]) {
        Fruit mango = new Mango(Color.YELLOW, true); // mango is seasonal
        Fruit banana = new Banana(Color.YELLOW, false); // banana is not seasonal

        List&lt;Fruit&gt; platter = new ArrayList&lt;Fruit&gt;();
        platter.add(mango);
        platter.add(banana);
        serve(platter);
    }

    public static void serve(Collection&lt;Fruit&gt; fruits) {
        System.out.println("Preparing fruits to serve");
        for (Fruit f : fruits) {
            f.prepare();
        }
    }
}
```

```java
abstract class Fruit {
    private Color color;
    private boolean seasonal;

    public Fruit(Color color, boolean seasonal) {
        this.color = color;
        this.seasonal = seasonal;
    }

   /*
    * This is an abstract method, see it doesn't have method body, only declaration
    */
    public abstract void prepare();

    public Color getColor() {
        return color;
    }

    public boolean isSeasonal() {
        return seasonal;
    }
}
```

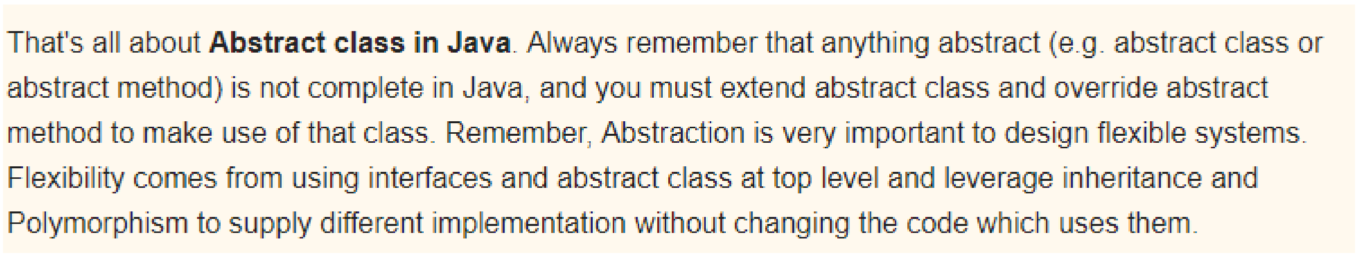
```java
class Mango extends Fruit {

    public Mango(Color color, boolean seasonal) {
        super(color, seasonal);
    }

    @Override
    public void prepare() {
        System.out.println("Cut the Mango");
    }
}
```

```
class Banana extends Fruit {

    public Banana(Color color, boolean seasonal) {
        super(color, seasonal);
        // TODO Auto-generated constructor stub
    }

    @Override
    public void prepare() {
        System.out.println("Peal the Banana");
    }
}
```

Output:

```
Preparing fruits to serve
Cut the Mango
Peal the Banana
```

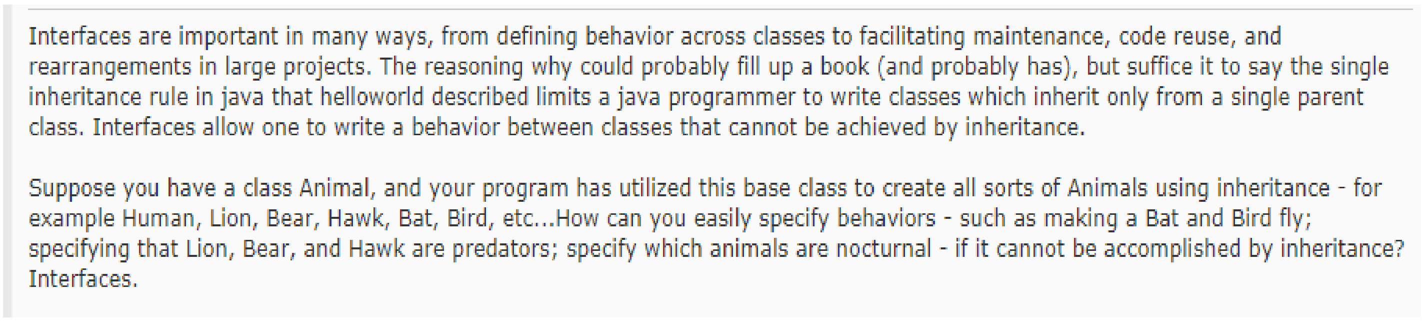
```
abstract class Fruit {
    private Color color;
    private boolean seasonal;

    public Fruit(Color color, boolean seasonal) {
        this.color = color;
        this.seasonal = seasonal;
    }

   /*
    * This is an abstract method, see it doesn't have method body, only declaration
    */
    public abstract void prepare();

    public Color getColor() {
        return color;
    }

    public boolean isSeasonal() {
        return seasonal;
    }
}
```



That's all about **Abstract class in Java**. Always remember that anything abstract (e.g. abstract class or abstract method) is not complete in Java, and you must extend abstract class and override abstract method to make use of that class. Remember, Abstraction is very important to design flexible systems. Flexibility comes from using interfaces and abstract class at top level and leverage inheritance and Polymorphism to supply different implementation without changing the code which uses them.

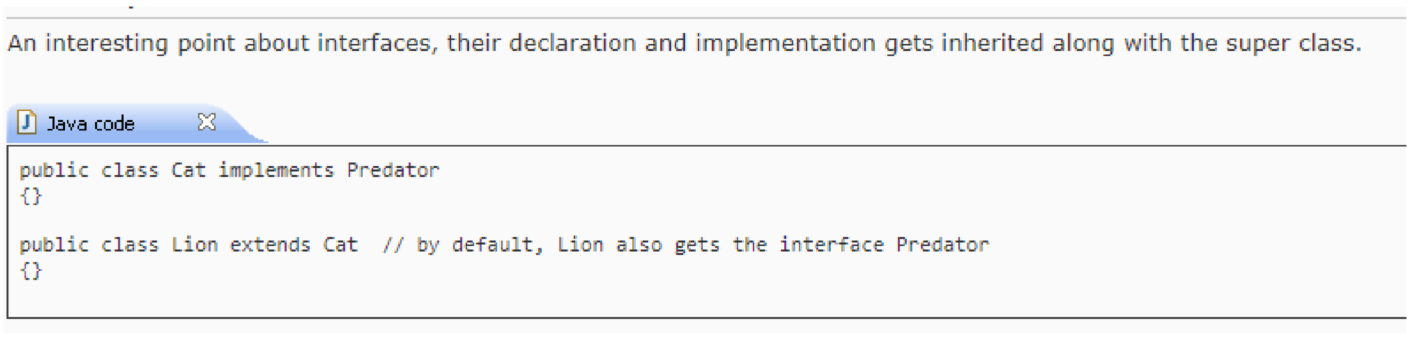
Interfaces:

As I see it, an interface is a way to gain polymorphic behavior without being limited by some languages' single-inheritance mechanism.



Interfaces are important in many ways, from defining behavior across classes to facilitating maintenance, code reuse, and rearrangements in large projects. The reasoning why could probably fill up a book (and probably has), but suffice it to say the single inheritance rule in java that helloworld described limits a java programmer to write classes which inherit only from a single parent class. Interfaces allow one to write a behavior between classes that cannot be achieved by inheritance.

Suppose you have a class Animal, and your program has utilized this base class to create all sorts of Animals using inheritance - for example Human, Lion, Bear, Hawk, Bat, Bird, etc...How can you easily specify behaviors - such as making a Bat and Bird fly; specifying that Lion, Bear, and Hawk are predators; specify which animals are nocturnal - if it cannot be accomplished by inheritance? Interfaces.



An interesting point about interfaces, their declaration and implementation gets inherited along with the super class.

```
public class Cat implements Predator
{}

public class Lion extends Cat  // by default, Lion also gets the interface Predator
{}
```