Abstract Methods in Abstract Class

```
abstract class Sum{
   //abstract methods
   public abstract int SumOfTwo(int n1, int n2);
   //Regular method
   public void disp(){System.out.println("Method of class Sum");}
}

class AbstractDemo extends Sum{
   public int SumOfTwo(int num1, int num2){return num1+num2;}
   public static void main(String args[]){
      AbstractDemo obj = new AbstractDemo();
      System.out.println(obj.SumOfTwo(3, 7));
      obj.disp();
   }
}
```

`Interface`

```
interface Animal {
   public void eat();  public void travel();
}

public class MammalInt implements Animal{
   public void eat(){ System.out.println("Mammal eats"); }
   public void travel(){ System.out.println("Mammal travels"); }
   public int noOfLegs(){ return 0;}
   public static void main(String args[]){
      MammalInt m = new MammalInt();
      m.eat();
      m.travel();
   }
}
```

`Difference between extends and implements`

`extends` is for ***extending*** a class. for Inheritance → **Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another**

`implements` is for ***implementing*** an interface

The difference between an interface and a regular class is that in an interface you can not implement any of the declared methods. Only the class that "implements" the interface can implement the methods. The C++ equivalent of an interface would be an abstract class (not EXACTLY the same but pretty much).

Also java doesn't support multiple inheritance for classes. This is solved by using multiple interfaces.

`Difference b/w interface and Abstract Class`

Abstract class	Interface
1) Abstract class can have abstract and non-abstractmethods.	**Interface can have only abstract methods.**
2) Abstract class doesn't support multiple inheritance.	**Interface supports multiple inheritance.**
3) Abstract class can have final, non-final, static and non-static variables.	**Interface has only static and final variables.**
4) Abstract class can have static methods, main method and constructor.	**Interface can't have static methods, main method or constructor.**
5) Abstract class can provide the implementation of interface.	**Interface can't provide the implementation of abstract class.**
6) The abstract keyword is used to declare abstract class.	**The interface keyword is used to declare interface.**
7) Example: **public abstract class Shape{** **public abstract void draw();** **}**	**Example:** **public interface Drawable{** **void draw();** **}**

`Polymorphism`

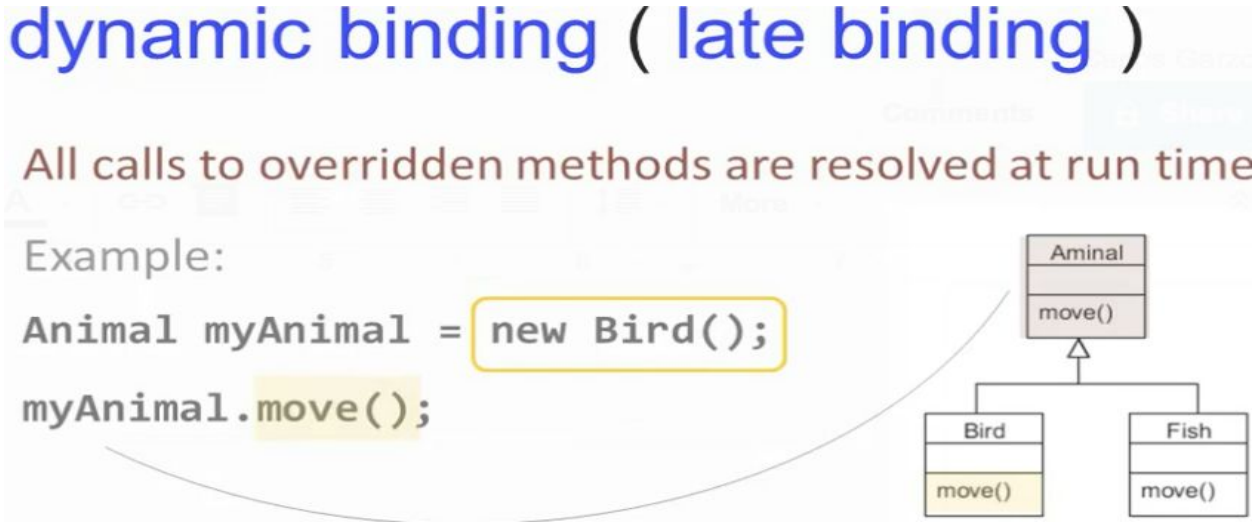
```
public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}

//The following is all legal

Deer d = new Deer();
Animal a = d;
Vegetarian v = d;
Object o = d;
```

Superclass/Subclass Assignments

super: ClassA	ClassA varA1 = varA; //good
↑	ClassA varA1 = varB; //good
Sub: ClassB	ClassB varB1 = varB; //good
	ClassB varB = (classB) varA ; //must be casted (“Downcasting”)

Examples

1. Animal a1 = new Bird();
   Bird myBird = (bird) a1; //works a1 is an instance of bird
2. Animal a2 = new Fish();
   Bird myBird = (Bird) a2; //fails a2 is NOT an isntance of Bird
3. Animal a3 = getsomeAnimal();
   Bird myBird = (Bird) a3; //might or might not work



a. In this case since myAnimal is a “Bird()” type it will use the method “move()” pertaining to bird not animal.

Static Binding

Normally, when you put a function call in your code, the compiler links it to the right function in the right class. The previous is “static binding”, however, for cases with interfaces real time information is not available at compile time.

Super

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
public class ChildClass extends BaseClass{
        public void printSomething(){ System.out.println(“This was printed from ChildClass”);}

        public static void main( String args[] )
        {
                ChildClass testSuper = new ChildClass();
                testSuper.printSomething();
        }
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
public class BaseClass{
        public void printSomething(){ System.out.println(“This was printed from BaseClass”);}
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

`Right now there are two` printSomething() methods and the ChildClass method will be used. “This was printed from ChildClass”. If we don’t want to change the name of the method and use the BaseClass printSomething() we use “Super” therefore →

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

public class ChildClass extends BaseClass{

        public void printSomething(){ super.printSomething();}

        public static void main( String args[] )
        {
                ChildClass testSuper = new ChildClass();
                testSuper.printSomething();
        }
}
```

Compiler now prints “This was printed from BaseClass”

Difference between overriding and overloading

Method **overloading** deals with the notion of having two or more methods (functions) in the same class with the same name but different arguments.**Method overloading** is defining several methods in the same class, that accept different numbers and types of parameters. In this case, the actual method called is decided at compile-time, based on the number and types of arguments. *For instance*, the method `System.out.println()` is overloaded, so that you can pass ints as well as Strings, and it will call a different version of the method.

Method **overriding** means having two methods with the same arguments, but different implementations. One of them would exist in the parent class (`superclass`) while another will be in the derived class (the child). The `@Override` annotation is required for this. See comments below. **Method overriding** is when a child class redefines the same method as a parent class, with the same parameters.

protected
protected methods are methods that are public to its' subclasses but private to anyone else