Today topic
  a. Dynamic Polymorphism
       1. Overriding
            (abstract class, interfaces)

  b. Static Polymorphism
      1. MethodHiding

Overloading
++++++++++++
 => Two or more methods with same name,but different argument type is refered as "Overloading".
 => In case of Overloading,Compiler will bind the methodcall based on the argument type we are passing, so
    we say Overloading has "FalsePolymorphism/EagerBinding/StaticBinding/EarlyBinding".

Overriding
++++++++++
 => During inheritance, the parent class method implementation would not match the needs of the child class so
    child class will take the method name,but it will change the implementation as per the needs of the child class
    This mechanism is called as "Overriding".
 => In case of Overriding, JVM will bind the method calls based on the runtime object, but not on the reference type so
    we say Overriding has "TruePolymorphism/LateBinding/RuntimeBinding".

eg#1
```
class Parent
{
      public void property(){
            System.out.println("Land+Cash+Gold");
      }

      public void marry(){
            System.out.println("RelativeGirl");
      }
}

class Child extends Parent
{
      //Overriding
      public void marry(){
            //Re-Implementation
            System.out.println("SomeOther Girl...");
      }
}
class Test {
      public static void main(String[] args) {

                  //Parent Object
                  Parent p1 = new Parent();
                  p1.property();
                  p1.marry();
```

```
                System.out.println();

                //Child Object
                Child c1 = new Child();
                c1.property();
                c1.marry();

                System.out.println();

                //Child Object
                Parent p2=new Child();
                p2.property();
                p2.marry();
        }
}
```

Output

```
D:\Decode Java1.0Batch>javac Test.java

D:\Decode Java1.0Batch>java Test
Land+Cash+Gold
RelativeGirl

Land+Cash+Gold
SomeOther Girl...

Land+Cash+Gold
SomeOther Girl...
```

eg#2.

```
class Plane
{
      public void takeOff(){
            System.out.println("Plane tookOff...");
      }

      public void fly(){
            System.out.println("Plane is flying...");
      }

      public void land(){
            System.out.println("Plane is landing...");
      }
}

class PassengerPlane extends Plane
{
      public void takeOff(){
            System.out.println("Passenger-Plane tookOff...");
      }
      public void fly(){
            System.out.println("Passenger-Plane is flying...");
      }

      public void land(){
            System.out.println("Passenger-Plane is landing...");
```

```
	}
}

class CargoPlane extends Plane
{
	public void takeOff(){
		System.out.println("Cargo-Plane tookOff...");
	}
	public void fly(){
		System.out.println("Cargo-Plane is flying...");
	}

	public void land(){
		System.out.println("Cargo-Plane is landing...");
	}
}

class FighterPlane extends Plane
{
	public void takeOff(){
		System.out.println("Fighter-Plane tookOff...");
	}
	public void fly(){
		System.out.println("Fighter-Plane is flying...");
	}

	public void land(){
		System.out.println("Fighter-Plane is landing...");
	}
}

class Airport
{
	//TruePolymorphism
	public void allowPlane(Plane p){
		p.takeOff();
		p.fly();
		p.land();
		System.out.println();
	}
}

class Test {
	public static void main(String[] args) {

		PassengerPlane p = new PassengerPlane();
		CargoPlane c= new CargoPlane();
		FighterPlane f =  new FighterPlane();

		Airport a= new Airport();
		a.allowPlane(p);
		a.allowPlane(c);
		a.allowPlane(f);

	}
}
Output
D:\Decode Java1.0Batch>javac Test.java
```

```
D:\Decode Java1.0Batch>java Test
Passenger-Plane tookOff...
Passenger-Plane is flying...
Passenger-Plane is landing...

Cargo-Plane tookOff...
Cargo-Plane is flying...
Cargo-Plane is landing...

Fighter-Plane tookOff...
Fighter-Plane is flying...
Fighter-Plane is landing...


eg#3.
class Animal
{
	public void eat(){
		System.out.println("Animal is Eating...");
	}
	public void sleep(){
		System.out.println("Animal is Sleeping...");
	}
}
class Monkey extends Animal
{
	public void eat(){
		System.out.println("Monkey steals and eats..");
	}
	public void sleep(){
		System.out.println("Monkey is Sleeping...");
	}
}
class Deer extends Animal
{
	public void eat(){
		System.out.println("Deer graze and eats...");
	}
	public void sleep(){
		System.out.println("Deer is Sleeping...");
	}
}
class Lion extends Animal
{
	public void eat(){
		System.out.println("Lion hunts and eats...");
	}
	public void sleep(){
		System.out.println("Lion is Sleeping...");
	}
}

class Forest
{
	public void allowAnimal(Animal  animal){
		animal.eat();
		animal.sleep();
		System.out.println();
	}
```

```
}

class Test {
	public static void main(String[] args) {

		Forest f =new Forest();
		f.allowAnimal(new Monkey());
		f.allowAnimal(new Deer());
		f.allowAnimal(new Lion());
	}
}
```

Output

```
D:\Decode Java1.0Batch>javac Test.java
D:\Decode Java1.0Batch>java Test
Monkey steals and eats..
Monkey is Sleeping...

Deer graze and eats...
Deer is Sleeping...

Lion hunts and eats...
Lion is Sleeping...
```

Rules of Overriding

++++++++++++++++++

1. In case of Overriding, we can't change the returntype of the method,if we want to change then there sould be relationship b.w
   returntype of the methods.

eg#1.

```
hclass Parent
{
	public Object methodOne(){
		return null;
	}
}
class Child extends Parent
{
	public void methodOne(){
		System.out.println("Hello from child...");
	}
}

public class Test {
	public static void main(String[] args) {
		Parent  p = new Child();
		p.methodOne();
	}
}
```

Output

```
CE: void and Object are not related.
```

eg#2.

```
class Parent
{
	public Object methodOne(){
		return null;
```

```
	}
}
class Child extends Parent
{
	public String methodOne(){
		System.out.println("Hello from child...");
		return null;
	}
}

public class Test {
	public static void main(String[] args) {
		Parent  p = new Child();
		p.methodOne();
	}
}
Output
Hello from child....


2. While overrding, we can't reduce the scope of access modifier.
	private< default < protected < public

eg#1.
class Parent
{
	public void methodOne(){
		System.out.println("Hello from Parent class...");
	}
}
class Child extends Parent
{
	protected void methodOne(){
		System.out.println("Hello from Child class...");
	}
}

public class Test {
	public static void main(String[] args) {
		Parent  p = new Child();
		p.methodOne();
	}
}
output
CE: can't reduce the scope of access modifier.

eg#2.
class Parent
{
	 void methodOne(){
		System.out.println("Hello from Parent class...");
	}
}
class Child extends Parent
{
	/protected/public void methodOne(){
		System.out.println("Hello from Child class...");
	}
}
```

```
public class Test {
	public static void main(String[] args) {
		Parent  p = new Child();
		p.methodOne();
	}
}
output
Hello from child class...
```

3.private methods won't participate in inheritance, so overriding them in child class is not possible.

```
class Parent
{
	private void methodOne(){
		System.out.println("Hello from Parent class...");
	}
}
class Child extends Parent
{
	private void methodOne(){
		System.out.println("Hello from Child class...");
	}
}
public class Test {
	public static void main(String[] args) {
		Parent  p = new Child();
		p.methodOne();
	}
}
Output
CE: error: methodOne() has private access in Parent
```

4. final is an access modifer applicable at
	a. variable => If applied at variable level, then the value can't be changed.
	b. method => If applied at method level, then we can't override the method in child class.
	c. class => If applied at class level, then the class won't participate in inheritance.

final methods can't be overriden in child class

```
eg#1.
class Parent
{
	public final void methodOne(){
		System.out.println("ParentClass:: methodOne()");
	}
}
class Child extends Parent
{
	public void methodOne(){
		System.out.println("ChildClass:: methodOne()");
	}
}
public class Test {
```

```
	public static void main(String[] args) {
		Parent  p = new Child();
		p.methodOne();
	}
}
Output
Test.java:9: error: methodOne() in Child cannot override methodOne() in Parent
	  public void methodOne(){
	              ^
  overridden method is final
1 error
```

5. abstract is an access modifer applicable at
	a. method -> If we are not giving the body for a method then mark the method as "abstract".
	b. class -> If we don't want the object to be created for a class, then mark the class as "abstract".
	c. variable-> This access modifier can't be applied on variables.

In case of overriding, compulsorily the child class should give implementation for all the abstract methods present in the parent class, if the implementation is not given then that child class should be marked as "abstract".

eg#1.

```
abstract class Parent
{
	public abstract void methodOne();
}
class Child extends Parent
{
	public void methodOne(){
		System.out.println("ChildClass:: methodOne()");
	}
}
public class Test {
	public static void main(String[] args) {
		Parent  p = new Child();
		p.methodOne();
	}
}
```

Output

```
ChildClass:: methodOne()
```