```
class Test
{
    public void accept(int d)
    {
        System.out.println("int");
    }
    public void accept(long s)
    {
        System.out.println("long");
    }
}
public class AmbiguityIssue {

    public static void main(String[] args)
    {
        Test t = new Test();
        t.accept(9);

    }
}
```

Note : Here int will be executed because int is the nearest type

---

```
class Test
{
    public void accept(Object s)
    {
        System.out.println("Object");
    }
    public void accept(String s)
    {
        System.out.println("String");
    }
}
public class AmbiguityIssue {

    public static void main(String[] args)
    {
        Test t = new Test();
        t.accept(9);
    }
}
```

Here Object will be executed.

```
class Test
{
    public void accept(Object s)
    {
        System.out.println("Object");
    }
    public void accept(String s)
    {
        System.out.println("String");
    }
}
public class AmbiguityIssue {

    public static void main(String[] args)
    {
        Test t = new Test();
        t.accept("hi");
    }
}
```

Here String will be executed

```
class Test
{
    public void accept(Object s)
    {
        System.out.println("Object");
    }
    public void accept(String s)
    {
        System.out.println("String");
    }
    public void accept(Integer i)
    {
        System.out.println("Integer");
    }
}
public class AmbiguityIssue {

    public static void main(String[] args)
    {
        Test t = new Test();
        t.accept(null);
    }
}
```

Here We will get compilation error

```
class Alpha
{
}
class Beta extends Alpha
{
}
class Test
{
    public void accept(Alpha s)
    {
        System.out.println("Alpha");
    }
    public void accept(Beta i)
    {
        System.out.println("Beta");
    }
}
public class AmbiguityIssue {

    public static void main(String[] args)
    {
        Test t = new Test();
        t.accept(null);
    }
}
```

Here Beta will be executed.

```
class Test
{
    public void accept(Number s)
    {
        System.out.println("Number");
    }
    public void accept(Integer i)
    {
        System.out.println("Integer");
    }
}
public class AmbiguityIssue {

    public static void main(String[] args)
    {
        Test t = new Test();
        t.accept(12);
    }
}
```

Here Integer will be executed.

```
class Test
{
    public void accept(long s)
    {
        System.out.println("Widening");
    }
    public void accept(Integer i)
    {
        System.out.println("Autoboxing");
    }
}
public class AmbiguityIssue
{
    public static void main(String[] args)
    {
        Test t = new Test();
        t.accept(12);
    }
}
```

Here widening is having more priority

```
class Test
{
    public void accept(int ...s)
    {
        System.out.println("Var args");
    }
    public void accept(Integer i)
    {
        System.out.println("Autoboxing");
    }
}
public class AmbiguityIssue {

    public static void main(String[] args)
    {
        Test t = new Test();
        t.accept(12);
    }
}
```

Here Autoboxing will be executed.

```
class Test
{
    public void accept(Number n)
    {
        System.out.println("Number");
    }
    public void accept(Double d)
    {
        System.out.println("Double");
    }
}
public class AmbiguityIssue {

    public static void main(String[] args)
    {
        Test t = new Test();
        t.accept(12);

    }
```

Here Number will be executed

```
class Test
{
    public void accept(int x, long y)
    {
        System.out.println("int-long");
    }
    public void accept(long x, int y)
    {
        System.out.println("long-int");
    }
}
public class AmbiguityIssue {

    public static void main(String[] args)
    {
        Test t = new Test();
        t.accept(9,1); //error
    }
}
```

What is Method Overriding ?

Method Overriding = Possible with **non static method**
Method Hiding = Possible with **Static Method**

```
class Super
{
  public void m1()
  {
  }
}
class Sub extends Super
{
  public void m1() //Method Overriding
  {
  }
}
```

Replacing super class method body implementation with sub class method is known as Overriding.

* If we write two or more than two non static methods in super and sub class with **same signature (Method Name + Method Parameter) and compatible return** type is called Method Overriding.

* Method Overriding is **not possible without Inheritance.**

* While working with Method Overriding, the return type of both the methods must be **compatible** so, Generally we **cannot** change the return type of the overridden method but from JDK 1.5V, We can change the return type of the Overridden Method by using **Co-Variant** (Same direction) concept.

What is the advantage of Method Overriding ?

The advantage of Method Overriding is, each sub class is specifying its own specific behavior as shown below. [Dog and Rabbit both are specifying their own behavior]

```
class Animal
{
  public void sleep()
  {
    System.out.println("Generic Animal is sleeping");
  }
}
class Dog extends Animal
{
  public void sleep()
  {
    System.out.println("Dog Animal is sleeping");
  }
}
class Rabbit extends Animal
{
  public void sleep()
  {
    System.out.println("Rabbit Animal is sleeping");
  }
}
```

Upcasting and Downcasting :

Upcasting :

* If we assign **sub class object to super class reference variable then It is called Upcasting.**

```
Animal a1 = new Dog();
```

Downcasting :

---

* Downcasting **is not possible without upcasting.**

* It is a technique to assign sub class object to sub class reference variable via super class reference variable.

* It is not possible in java to assign super class object to sub class class reference variable, otherwise we will get java.lang.ClassCastException.

Case 1 :

```
Animal a = new Dog();   [Valid It is upcasting]
```

Case 2 :

```
Dog d = new Animal();  //Compilation error
```

Case 3 :

```
Dog d = (Dog) new Animal();
```

Here we don't have compilation error but we will get java.lang.ClassCastException

Case 4 :

```
Animal a1 = new Dog(); //upcasting

Dog d1 = (Dog) a1; //Downcasting
```

Note : Downcasting is not possible without Upcasting

Method Execution in Upcasting :

Example :

```
class Animal
{
  public void sleep()
  {
    System.out.println("Generic Animmal");
  }
}
class Dog extends Animal
{
  public void sleep()
  {
    System.out.println("Dog Animmal ");
  }
}
```

Animal a1 = new Dog();
a1.sleep();

Compiler Activity :

Here a1 variable is of type **Animal** so, compiler will search sleep() method in Animal class.

JVM Activity :

JVM will see, object is creted for **Dog class** so It will **start executing from Dog class [Bottom to top]**