Method Overloading And Method Overriding

Method overloading

1. When a class has more than one method having the same name but with different parameter lists, this feature is called method overloading in Java.
2. Overloading is one of the ways that Java implements polymorphism.
3. It is a powerful and very useful feature for increasing the maintainability and readability of code.
4. Java compiler differentiates overloaded methods with their signatures. The signature of a method is defined by its name and a list of parameters.
5. The return type of method is not part of the method signature. It does not play any role in resolving methods overloaded.

Method Overloading Rules in Java

Below are the list of rules by which we can implement method overloading in Java.

1. The method name must be the same.
2. Parameters must be different, i.e. each overloaded method must take a unique list of parameter types. We can change the parameters in one of the following three ways:

   a) **Data type of parameters**

   b) **Number of parameters**

   c) **Sequence of data type of parameters**

Features of Method overloading

1. The call to overloaded method is bonded at compile time.
2. The concept of method overloading is also known as compile-time polymorphism in java.
3. Method overloading is generally done in the same class. But we can also do it in the subclass.
4. Method overloading in Java cannot be done by changing the return type of the method because there may occur ambiguity.
5. We can overload the private methods in Java.

Features of Method overloading

6. The final methods can be overloaded in Java.
7. The main method can also be overloaded in Java.
8. We can overload both static and instance methods in Java.
9. Method overloading is possible when two or more static methods with the same name, but the difference in the list of parameters.
10. We can stop method overriding by declaring method as **final**.

Method Overriding in Java

1. Method overriding in Java means redefining a method in a subclass to replace the functionality of superclass method.
2. When the method of superclass is overridden in the subclass to provide more specific implementation, it is called method overriding.
3. If you are not satisfied with the implementation (or functionality or behavior) of an inherited method, you do not need to modify that method in superclass.
4. To override a method in a subclass, the method must be defined in the subclass using the same signature and same return type as in its superclass.
5. The superclass method, which is overridden, is called overridden method in Java.
6. The subclass method which is overriding the superclass method is called overriding method in Java.

Features of Method Overriding

1. Method overriding technique supports the runtime polymorphism.
2. It allows a subclass to provide its own implementation of the method which is already provided by the superclass.
3. Only the instance method can be overridden in Java.
4. An instance variable can never be overridden in Java.
5. Overriding concept is not applicable for private, final, static, and main method in Java.
6. Similarly, the default method of superclass can be overridden by default, protected, or public.
7. We cannot override a method if we do not inherit it. A private method cannot be overridden because it cannot be inherited in the subclass.

@Override Annotation in Java

@Override is an annotation that was introduced in Java version 5.

It is used by the compiler to check all the rules of overriding.

If we do not write this annotation, the compiler will apply only three overriding rules such as:

1.) Superclass and subclass relation.

2.) Method name same.

3.) Parameters are the same.

But if we have written as **@Override** on subclass method, the compiler will apply all rules irrespective of the above three points.

Overloading vs Overriding

SN	Property	Overloading	Overriding
1	**Argument type**	Must be different (at least order).	Must be the same (including order).
2	**Method signatures**	Must be different.	Must be the same.
3	**Return type**	Same or different.	Must be the same until Java 1.4 version only. Java 1.5 onwards, Covariant return type is allowed.
4	**Class**	Generally performed in the same class.	Performed in two classes through Inheritance (Is-A relationship).
5	**Private/Static/Final method**	Can be overloaded.	Cannot be overridden.

Overloading vs Overriding

SN	Property	Overloading	Overriding
6	**Throws clause**	Anything	If child class method throws any checked exception compulsory parent class method should throw the same exception is its parent otherwise we will get compile-time error but there is no restriction for an unchecked exception.
7	**Method resolution**	Always take care by Java compiler based on reference type.	Always take care by JVM based on runtime object.
8	**Polymorphism**	Also known as compile-time polymorphism, static polymorphism, or early binding.	Also known as runtime polymorphism, dynamic polymorphism, or late binding.

Method Hiding in Java

- A static method (class method) cannot be overridden in Java.
- But if a static method defined in the parent class is redefined in a child class, the child class's method hides the method defined in the parent class.
- This mechanism is called method hiding in Java or function hiding.

Rules and Features of Method Hiding in Java

1. Both parent and child class methods must be static.
2. Method hiding is also known as compile-time polymorphism because the compiler is responsible to resolve method resolution based on the reference type.
3. It is also known as static polymorphism or early binding.
4. In method hiding, method call is always resolved by Java compiler based on the reference type. There is no role of runtime polymorphism in method hiding in Java.
5. The use of static in method declaration must be the same between superclass and subclass.

Method Hiding Vs Method Overriding

1. In method hiding, both parent and child class methods should be static whereas, in overriding, both parent and child class methods should be non-static.

2. Compiler is responsible for method resolution based on reference type whereas, in method overriding, JVM is always responsible for method resolution based on runtime object.

3. Method hiding is also known as compile-time polymorphism, static polymorphism, or early binding whereas, method overriding is also known as runtime polymorphism, dynamic polymorphism, or late binding.

Hiding Variable in Java

- When a variable defined in the parent class is redefined with the same name in a child class, the child class's variable hides variable defined in the parent class.
- This mechanism is called **variable hiding** in Java.
- Variable hiding is useful when you want to reuse the same variable name in the subclass.

Type Conversion in Java | Type Casting

The process of converting a value from one data type to another is known as type conversion in Java. Type conversion is also known as type casting in Java or simply '**casting**'.

If two data types are compatible with each other, Java will perform such conversion automatically or implicitly for you. We can easily convert a primitive data type into another primitive data type by using type casting.

Similarly, it is also possible to convert a non-primitive data type (referenced data type) into another non-primitive data type by using type casting.

Type Conversion in Java | Type Casting

But we cannot convert a primitive data type into an advanced (referenced) data type by using type casting.

For this case, we will have to use methods of Java Wrapper classes.

Types of Casting in Java

Two types of casting are possible in Java are as follows:

1. **Implicit type casting** (also known as automatic type conversion)
2. **Explicit type casting**

Implicit Type Casting in Java

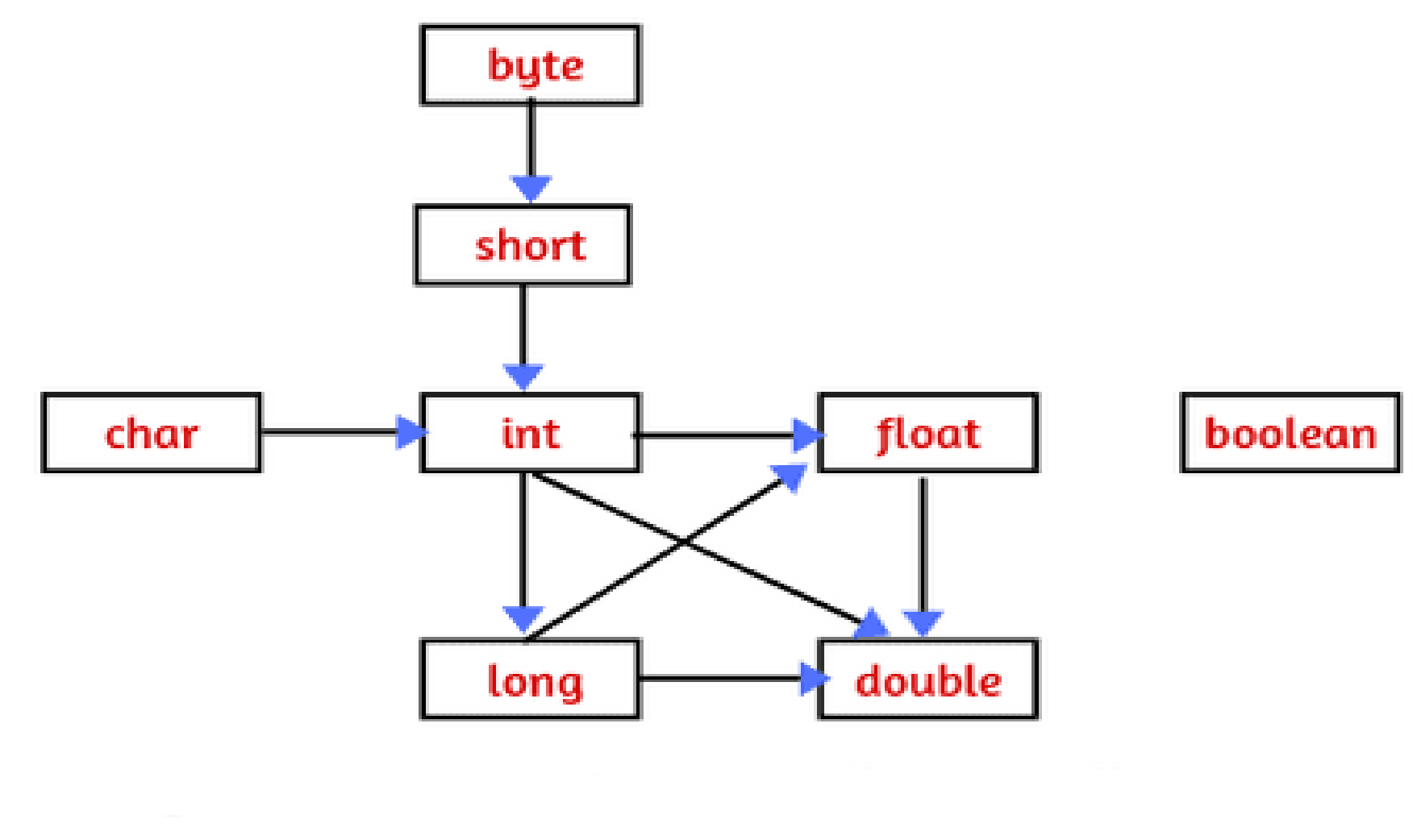
Automatic conversion (**casting**) done by Java compiler internally is called **implicit** conversion or implicit type casting in java.

Implicit casting is performed to convert a lower data type into a higher data type. It is also known as automatic type promotion in Java.

For example, if we assign an **int** value to a **long** variable, it is compatible with each other, but an **int** value cannot be assigned to a **byte** variable.

```
int x = 20;
long y = x;  // Automatic conversion
```

Implicit Type Casting in Java



Automatic Type Promotion Rules in Expression

1. If **byte**, **short**, and **int** are used in a mathematical expression, Java always converts the result into an int.
2. If a single **long** is used in the expression, the whole expression is converted to **long**.
3. If a **float** operand is used in an expression, the whole expression is converted to **float**.
4. If any operand is **double**, the result is promoted to double.
5. **Boolean** values cannot be converted to another type.

Automatic Type Promotion Rules in Expression

6. Conversion from **float** to **int** causes truncation of the fractional part which represents the loss of precision. Java does not allow this.

7. Conversion from **double** to **float** causes rounding of digits that may cause some of the value's precision to be lost.

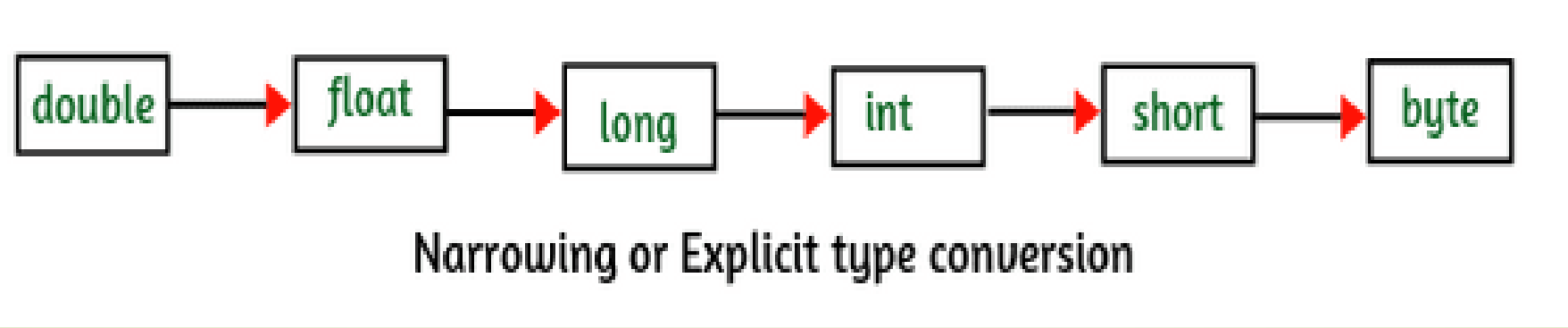
8. Conversion from **long** to **int** is also not possible. It causes the dropping of the excess higher order bits.

Explicit Type Casting (Narrowing Conversion) in Java

The conversion of a higher data type into a lower data type is called narrowing conversion.

Since this type of conversion is performed by the programmer, not by the compiler automatically, therefore, it is also called explicit type casting in Java. It is done to convert from a higher data type to a lower data type.

The following diagram is useful to perform the narrowing conversion or explicit type casting in Java program.



Explicit Type Casting (Narrowing Conversion) in Java

To perform explicit type casting, we will have to use a cast operator.

A cast operator is used to cast a primitive value from one type to another.

Using a cast operator is simply an explicit type conversion.

Syntax:

(type_name) expression;

Example:

```
int x;
   double  y =  9.99;
   x = (int)y;    // It will compile in Java and the resulting value will simply be 9.
```

Explicit Type Casting (Narrowing Conversion) in Java

```
double d = 100.9;
long l = (long)d;        // Explicit type casting.
// The output will be 100 because the fractional part is lost.


int x;
   float y = 27.6f;
   x = (int)(y + 0.5);
// On explicit type casting, the output becomes 28.


int a = 66;
char ch = (char)a; // ch stores 'B'.
```

Disadvantage of Explicit Type Casting in Java

There are the following disadvantages of using type casting in Java.

1. When you will use type casting in Java, you can lose some information or data.

2. Accuracy can be lost while using type casting.

3. When a double is cast to an int, the fractional part of a double is discarded, which causes the loss of fractional part of data.

Difference between Implicit Casting and Explicit Casting in Java

1. Implicit type casting is done internally by Java compiler whereas, explicit type casting is done by the programmer. Java compiler does not perform it automatically.

2. In explicit casting, cast operator is needed whereas, no need for any operator needs in the case of implicit type casting.

3. If we perform explicit type casting in a program, we can lose information or data, but in the case of implicit type casting, there is no loss of data.

4. Accuracy is not maintained in explicit type casting whereas, there is no issue of accuracy in implicit type conversion.

5. Implicit type conversion is safe, but explicit type casting is not safe.

Class Casting in Java | Generalization, Specialization

Class casting in Java

The process of converting a class type into another class type having the relationship between them through inheritance is called **class casting** in Java.

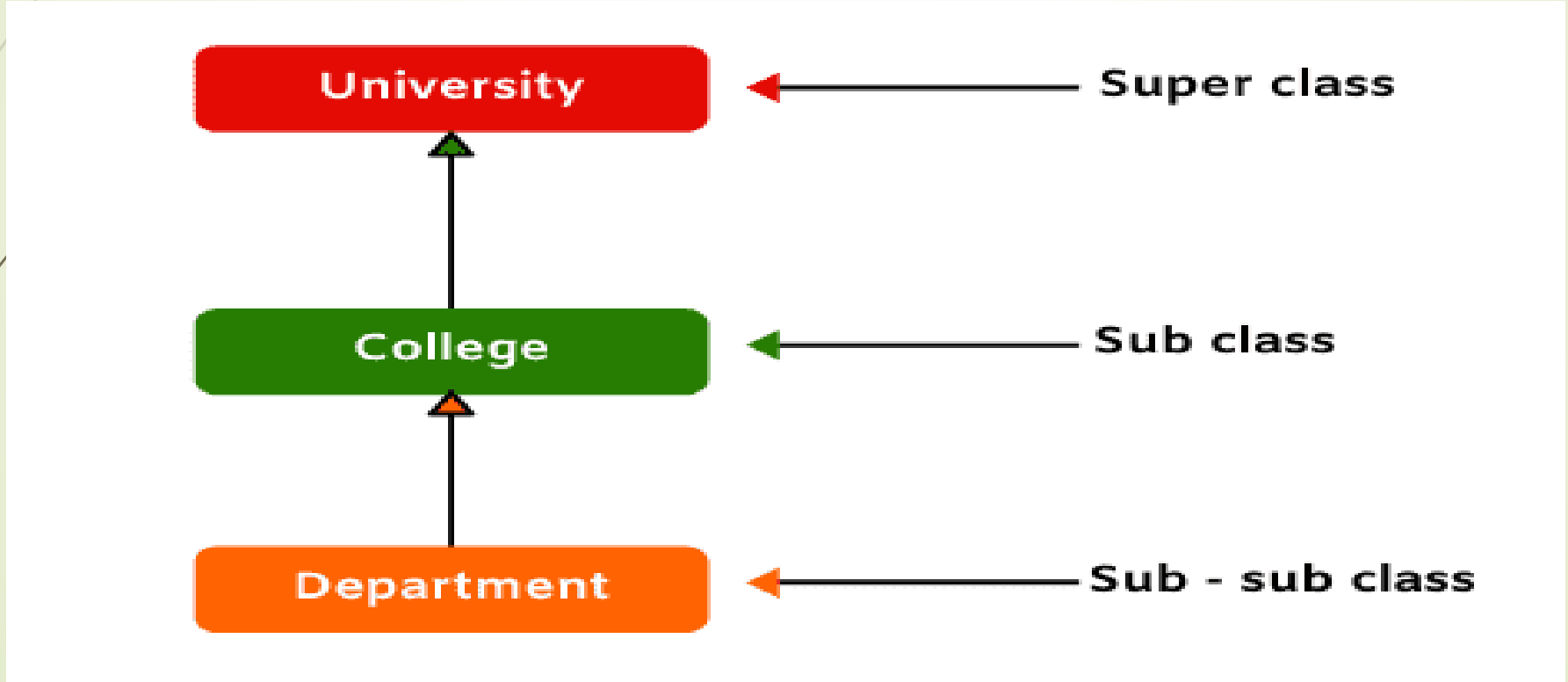
We have known in the previous tutorial that a class is a referenced data type.

If classes have some relationship between them through inheritance, it is also possible to convert a class type into another class type through type casting.

For example, we can not convert a Dog class into Animal class if there is no relationship between them through inheritance.

Class casting in Java

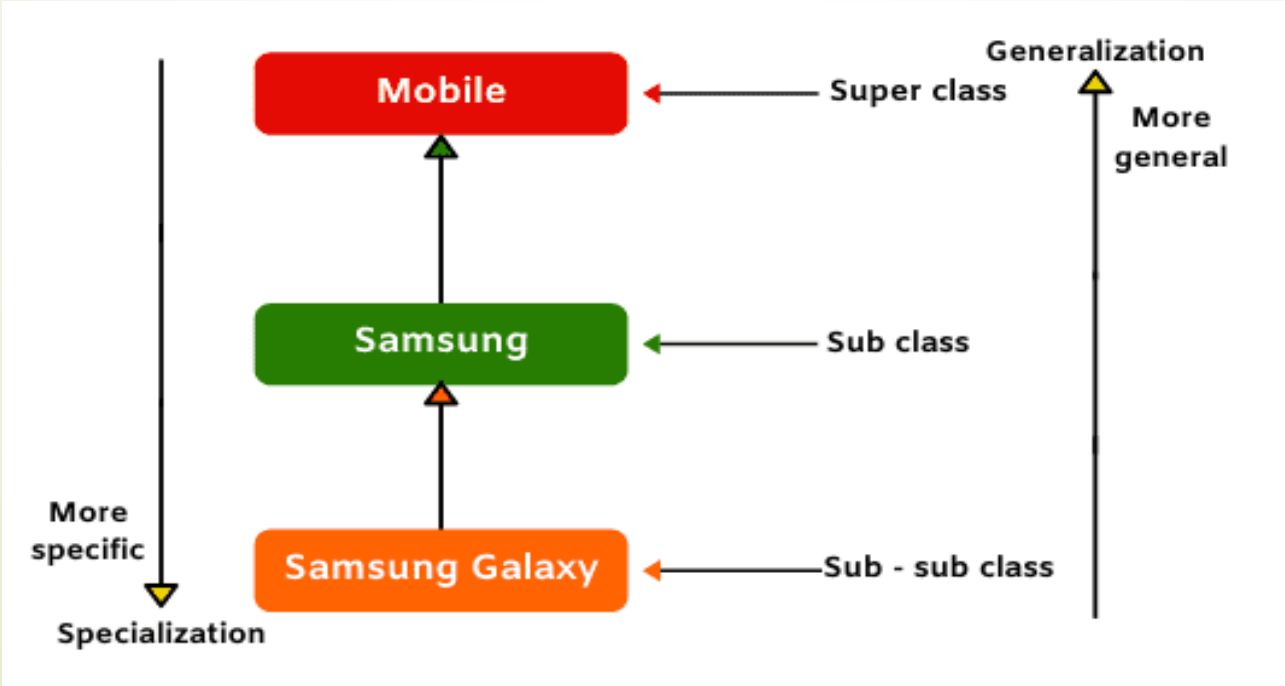
Examples of class casting in java.



Generalization in Java

The process of converting subclass type into superclass type is called **Generalization** in Java. This is because we are making the subclass to become more general so that its scope can be more widening.

This conversion is also called **widening** or **upcasting** in referenced data types.



Example of Generalization:

```
public class A {
 void m1() {
    System.out.println("Superclass
method");
 }
}
public class B extends A
{
 void m2()
{
    System.out.println("Subclass
method");
 }
}
```

```
public class WideningTest {
public static void main(String[]
args)
{
   A   a;   // a is a superclass
reference variable.
   a = new  B(); // generalization
(widening) because a is
referring to a subclass object.
   a.m1();
 }
}
```

Key Points:

1. If superclass reference refers to subclass object, all the methods of superclass is accessible but not subclass method.

2. If subclass reference refers to subclass object, all the methods of superclass and subclass are accessible because subclass objects avails a copy of superclass.

3. Generalization (widening) is performed by using subclass object, only methods of superclass are accessible. If we override methods of superclass into subclass, only subclass methods are accessible.

4. If specialization (narrowing) is performed by using superclass object, none of the superclass or subclass methods are accessible. It is useless.

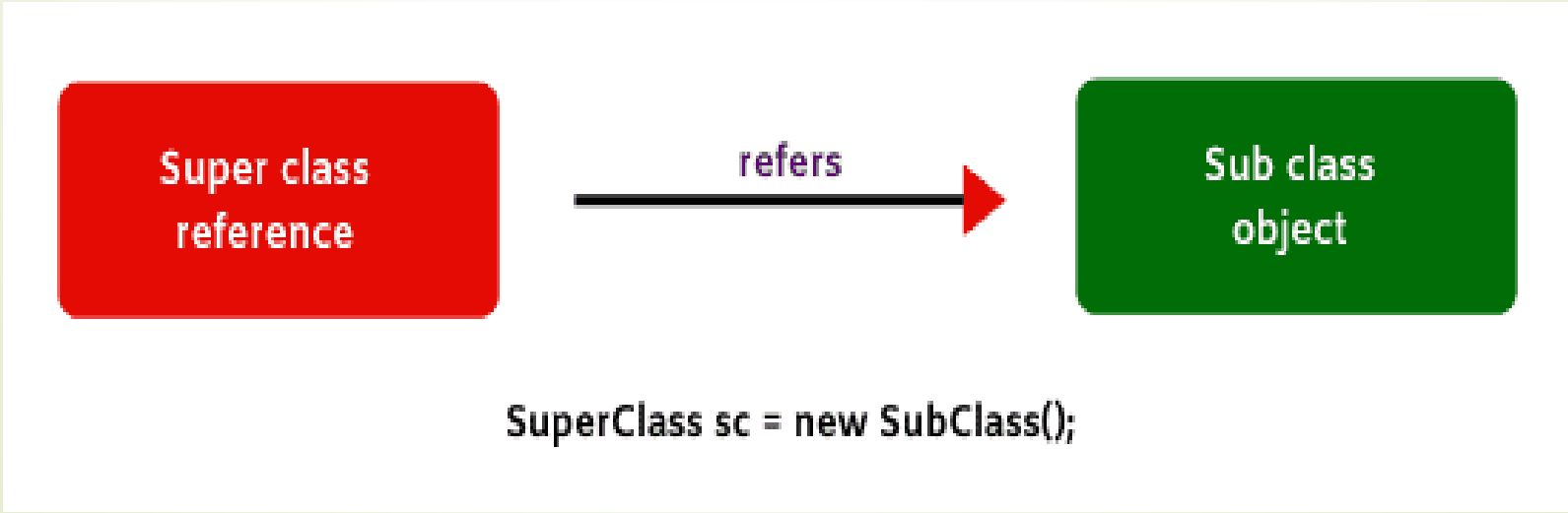
5. If narrowing is performed by using subclass object, all the methods of superclass and subclasses can be accessed.

Upcasting and Downcasting in Java

Upcasting in Java

When the reference variable of superclass refers to the object of subclass, it is known as **upcasting** in Java.

In other words, when the subclass object type is converted into the superclass type, this type of conversion is called **upcasting**. It is also called widening in Java.



Downcasting (Narrowing) in Java

When subclass reference refers to superclass object, it is called **downcasting** in Java.

In other words, when subclass type is converted into superclass type, this type of conversion is called **downcasting**.

It is also called **narrowing** in Java.