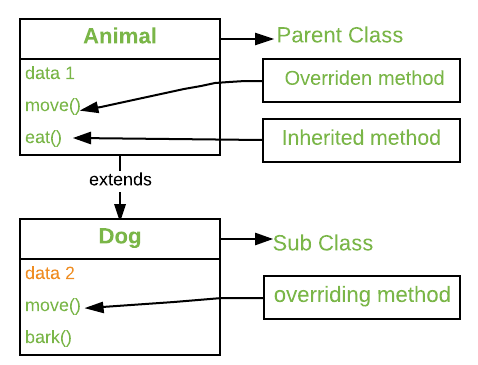
In any object-oriented programming language, Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature, and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.



Method overriding is one of the way by which java achieve [Run Time Polymorphism](https://www.geeksforgeeks.org/dynamic-method-dispatch-runtime-polymorphism-java/).The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed. In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.

//A Simple Java program to demonstrate

//method overriding in java

//Base Class

**class** Parent {

**void** show() {

System.***out***.println("Parent's show()");

}

}

//Inherited class

**class** Child **extends** Parent {

// This method overrides show() of Parent

@Override

**void** show() {

System.***out***.println("Child's show()");

}

}

//Driver class

**public** **class** Main {

**public** **static** **void** main(String[] args) {

// If a Parent type reference refers

// to a Parent object, then Parent's

// show is called

Parent obj1 = **new** Parent();

obj1.show();

// If a Parent type reference refers

// to a Child object Child's show()

// is called. This is called RUN TIME

// POLYMORPHISM.

Parent obj2 = **new** Child();

obj2.show();

}

}

**Rules for method overriding:**

1. **Overriding and Access-Modifiers :**The [access modifier](https://www.geeksforgeeks.org/access-modifiers-java/) for an overriding method can allow more, but not less, access than the overridden method. For example, a protected instance method in the super-class can be made public, but not private, in the subclass. Doing so, will generate compile-time error.

**class** Parent {

// private methods are not overridden

**private** **void** m1() {

System.***out***.println("From parent m1()");

}

**protected** **void** m2() {

System.***out***.println("From parent m2()");

}

}

**class** Child **extends** Parent {

// new m1() method

// unique to Child class

**private** **void** m1() {

System.***out***.println("From child m1()");

}

// overriding method

// with more accessibility

@Override

**public** **void** m2() {

System.***out***.println("From child m2()");

}

}

// Driver class

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Parent obj1 = **new** Parent();

obj1.m2();

Parent obj2 = **new** Child();

obj2.m2();

}

}

1. **Final methods can not be overridden :**If we don’t want a method to be overridden, we declare it as [final](https://www.geeksforgeeks.org/final-keyword-java/). Please see [Using final with Inheritance](https://www.geeksforgeeks.org/using-final-with-inheritance-in-java/).

**class** Parent {

// Can't be overridden

**final** **void** show() {

}

}

**class** Child **extends** Parent {

// This would produce error

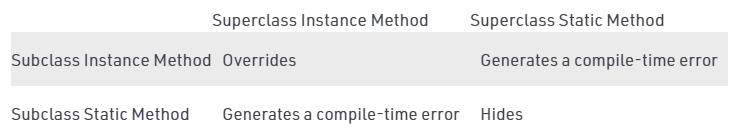
**void** show() { // error Cannot override the final method from Parent

}

}

1. **Static methods can not be overridden(Method Overriding vs Method Hiding) :**When you define a static method with same signature as a static method in base class, it is known as [method hiding](https://www.geeksforgeeks.org/can-we-overload-or-override-static-methods-in-java/).

The following table summarizes what happens when you define a method with the same signature as a method in a super-class.



//A Java program to demonstrate that

//final methods cannot be overridden

//Java program to show that

//if the static method is redefined by

//a derived class, then it is not

//overriding, it is hiding

**class** Parent {

// Static method in base class

// which will be hidden in subclass

**static** **void** m1() {

System.***out***.println("From parent " + "static m1()");

}

// Non-static method which will

// be overridden in derived class

**void** m2() {

System.***out***.println("From parent " + "non-static(instance) m2()");

}

}

**class** Child **extends** Parent {

// This method hides m1() in Parent

**static** **void** m1() {

System.***out***.println("From child static m1()");

}

// This method overrides m2() in Parent

@Override

**public** **void** m2() {

System.***out***.println("From child " + "non-static(instance) m2()");

}

}

//Driver class

**class** Main {

**public** **static** **void** main(String[] args) {

Parent obj1 = **new** Child();

// As per overriding rules this

// should call to class Child static

// overridden method. Since static

// method can not be overridden, it

// calls Parent's m1()

obj1.*m1*();

// Here overriding works

// and Child's m2() is called

obj1.m2();

}

}

1. **Private methods can not be overridden :**[Private methods](https://www.geeksforgeeks.org/can-override-private-methods-java/)cannot be overridden as they are bonded during compile time. Therefore we can’t even override private methods in a subclass.(See [this](https://www.geeksforgeeks.org/can-override-private-methods-java/) for details).

1. **The overriding method must have same return type (or subtype) :**From Java 5.0 onwards it is possible to have different return type for a overriding method in child class, but child’s return type should be sub-type of parent’s return type. This phenomena is known as [**covariant return type**](https://www.geeksforgeeks.org/covariant-return-types-java/).

1. **Invoking overridden method from sub-class :**We can call parent class method in overriding method using [super keyword](http://quiz.geeksforgeeks.org/super-keyword/).

//A Java program to demonstrate that overridden

//method can be called from sub-class

//Base Class

**class** Parent {

**void** show() {

System.***out***.println("Parent's show()");

}

}

//Inherited class

**class** Child **extends** Parent {

// This method overrides show() of Parent

@Override

**void** show() {

**super**.show();

System.***out***.println("Child's show()");

}

}

//Driver class

**class** Main {

**public** **static** **void** main(String[] args) {

Parent obj = **new** Child();

obj.show();

}

}

1. **Overriding and constructor :**We can not override constructor as parent and child class can never have constructor with same name(Constructor name must always be same as Class name).
2. **Overriding and Exception-Handling :**Below are two rules to note when overriding methods related to exception-handling.

* **Rule#1 :** If the super-class overridden method does not throw an exception, subclass overriding method can only throws the [unchecked exception](https://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/), throwing checked exception will lead to compile-time error.

**class** Parent {

**void** m1() {

System.***out***.println("From parent m1()");

}

**void** m2() {

System.***out***.println("From parent m2()");

}

}

**class** Child **extends** Parent {

@Override

// no issue while throwing unchecked exception

**void** m1() **throws** ArithmeticException {

System.***out***.println("From child m1()");

}

@Override

// compile-time error

// Exception Exception is not compatible with throws clause in Parent.m2()

**void** m2() **throws** Exception {

System.***out***.println("From child m2");

}

}

**Rule#2 :** If the super-class overridden method does throws an exception, subclass overriding method can only throw same, subclass exception. Throwing parent exception in [Exception hierarchy](https://www.geeksforgeeks.org/exceptions-in-java/) will lead to compile time error.Also there is no issue if subclass overridden method is not throwing any exception.

//Java program to demonstrate overriding when

//superclass method does declare an exception

**class** Parent {

**void** m1() **throws** RuntimeException {

System.***out***.println("From parent m1()");

}

}

**class** Child1 **extends** Parent {

@Override

// no issue while throwing same exception

**void** m1() **throws** RuntimeException {

System.***out***.println("From child1 m1()");

}

}

**class** Child2 **extends** Parent {

@Override

// no issue while throwing subclass exception

**void** m1() **throws** ArithmeticException {

System.***out***.println("From child2 m1()");

}

}

**class** Child3 **extends** Parent {

@Override

// no issue while not throwing any exception

**void** m1() {

System.***out***.println("From child3 m1()");

}

}

**class** Child4 **extends** Parent {

@Override

// compile-time error Exception Exception is not compatible with throws clause in Parent.m1()

// issue while throwing parent exception

**void** m1() **throws** Exception {

System.***out***.println("From child4 m1()");

}

}

1. **Overriding and abstract method:**Abstract methods in an interface or abstract class are meant to be overridden in derived concrete classes otherwise a compile-time error will be thrown.

1. **Overriding and synchronized/strictfp method :**The presence of synchronized/strictfp modifier with method have no effect on the rules of overriding, i.e. it’s possible that a synchronized/strictfp method can override a non synchronized/strictfp one and vice-versa.

**Note :**

* 1. In C++, we need[virtual keyword](https://www.geeksforgeeks.org/virtual-functions-and-runtime-polymorphism-in-c-set-1-introduction/) to achieve overriding or [Run Time Polymorphism](https://www.geeksforgeeks.org/dynamic-method-dispatch-runtime-polymorphism-java/). In Java, methods are virtual by default.
  2. We can have multilevel method-overriding.

//A Java program to demonstrate

//multi-level overriding

//Base Class

**class** Parent {

**void** show() {

System.***out***.println("Parent's show()");

}

}

//Inherited class

**class** Child **extends** Parent {

// This method overrides show() of Parent

**void** show() {

System.***out***.println("Child's show()");

}

}

//Inherited class

**class** GrandChild **extends** Child {

// This method overrides show() of Parent

**void** show() {

System.***out***.println("GrandChild's show()");

}

}

//Driver class

**class** Main {

**public** **static** **void** main(String[] args) {

Parent obj1 = **new** GrandChild();

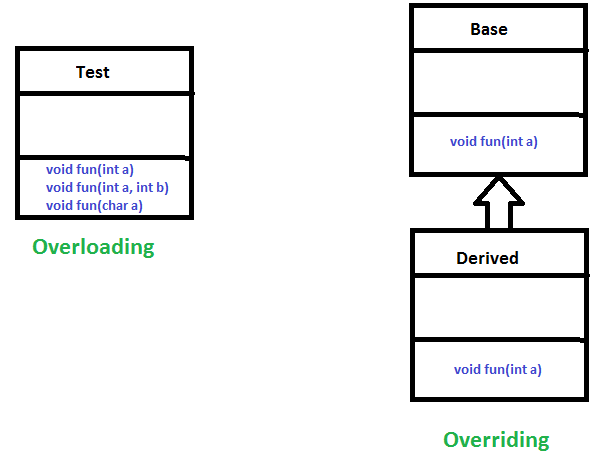
obj1.show();

}

}

**Overriding vs**[Overloading](https://www.geeksforgeeks.org/overloading-in-java/)**:**

* 1. Overloading is about same method have different signatures. Overriding is about same method, same signature but different classes connected through inheritance.

[](https://media.geeksforgeeks.org/wp-content/uploads/OverridingVsOverloading.png)

* 1. Overloading is an example of compiler-time polymorphism and overriding is an example of [run time polymorphism](https://www.geeksforgeeks.org/dynamic-method-dispatch-runtime-polymorphism-java/).

**Why Method Overriding ?**

As stated earlier, overridden methods allow Java to support [run-time polymorphism](https://www.geeksforgeeks.org/dynamic-method-dispatch-runtime-polymorphism-java/). Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives while allowing subclasses to define the specific implementation of some or all of those methods. Overridden methods are another way that Java implements the “one interface, multiple methods” aspect of polymorphism.

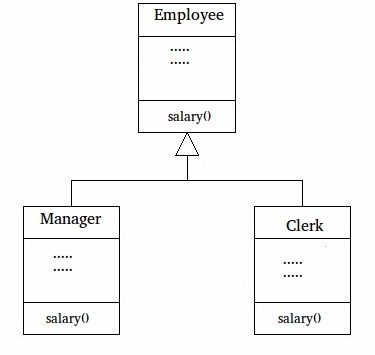
[Dynamic Method Dispatch](https://www.geeksforgeeks.org/dynamic-method-dispatch-runtime-polymorphism-java/) is one of the most powerful mechanisms that object-oriented design brings to bear on code reuse and robustness. The ability to exist code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.

Overridden methods allow us to call methods of any of the derived classes without even knowing the type of derived class object.

**When to apply Method Overriding ?(with example)**

**Overriding and**[Inheritance](https://www.geeksforgeeks.org/inheritance-in-java/) : Part of the key to successfully applying polymorphism is understanding that the superclasses and subclasses form a hierarchy which moves from lesser to greater specialization. Used correctly, the superclass provides all elements that a subclass can use directly. It also defines those methods that the derived class must implement on its own. This allows the subclass the flexibility to define its methods, yet still enforces a consistent interface. **Thus, by combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses.**

Let’s look at a more practical example that uses method overriding. Consider an employee management software for an organization, let the code has a simple base class Employee, the class has methods like raiseSalary(), transfer(), promote(), .. etc. Different types of employees like Manager, Engineer, ..etc may have their implementations of the methods present in base class Employee. In our complete software, we just need to pass a list of employees everywhere and call appropriate methods without even knowing the type of employee. For example, we can easily raise the salary of all employees by iterating through the list of employees. Every type of employee may have its logic in its class, we don’t need to worry because if raiseSalary() is present for a specific employee type, only that method would be called.

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/JavaOVerriding.jpg)

//A Simple Java program to demonstrate application

//of overriding in Java

//Base Class

**class** Employee {

**public** **static** **int** *base* = 10000;

**int** salary() {

**return** *base*;

}

}

//Inherited class

**class** Manager **extends** Employee {

// This method overrides salary() of Parent

**int** salary() {

**return** *base* + 20000;

}

}

//Inherited class

**class** Clerk **extends** Employee {

// This method overrides salary() of Parent

**int** salary() {

**return** *base* + 10000;

}

}

//Driver class

**class** Main {

// This method can be used to print the salary of

// any type of employee using base class reference

**static** **void** printSalary(Employee e) {

System.***out***.println(e.salary());

}

**public** **static** **void** main(String[] args) {

Employee obj1 = **new** Manager();

// We could also get type of employee using

// one more overridden method.loke getType()

System.***out***.print("Manager's salary : ");

*printSalary*(obj1);

Employee obj2 = **new** Clerk();

System.***out***.print("Clerk's salary : ");

*printSalary*(obj2);

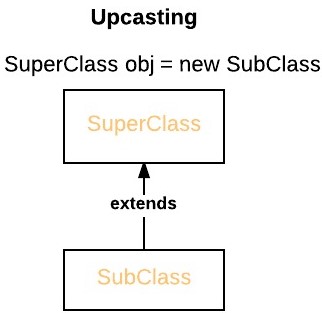
}

}

**Dynamic Method Dispatch or Runtime Polymorphism in Java**

Method overriding is one of the ways in which Java supports Runtime Polymorphism. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

* When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.
* At run-time, it depends on the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed
* A superclass reference variable can refer to a subclass object. This is also known as upcasting. Java uses this fact to resolve calls to overridden methods at run time.



Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed. Here is an example that illustrates dynamic method dispatch:

//A Java program to illustrate Dynamic Method

//Dispatch using hierarchical inheritance

**class** A {

**void** m1() {

System.***out***.println("Inside A's m1 method");

}

}

**class** B **extends** A {

// overriding m1()

**void** m1() {

System.***out***.println("Inside B's m1 method");

}

}

**class** C **extends** A {

// overriding m1()

**void** m1() {

System.***out***.println("Inside C's m1 method");

}

}

//Driver class

**public** **class** Main {

**public** **static** **void** main(String args[]) {

// object of type A

A a = **new** A();

// object of type B

B b = **new** B();

// object of type C

C c = **new** C();

// obtain a reference of type A

A ref;

// ref refers to an A object

ref = a;

// calling A's version of m1()

ref.m1();

// now ref refers to a B object

ref = b;

// calling B's version of m1()

ref.m1();

// now ref refers to a C object

ref = c;

// calling C's version of m1()

ref.m1();

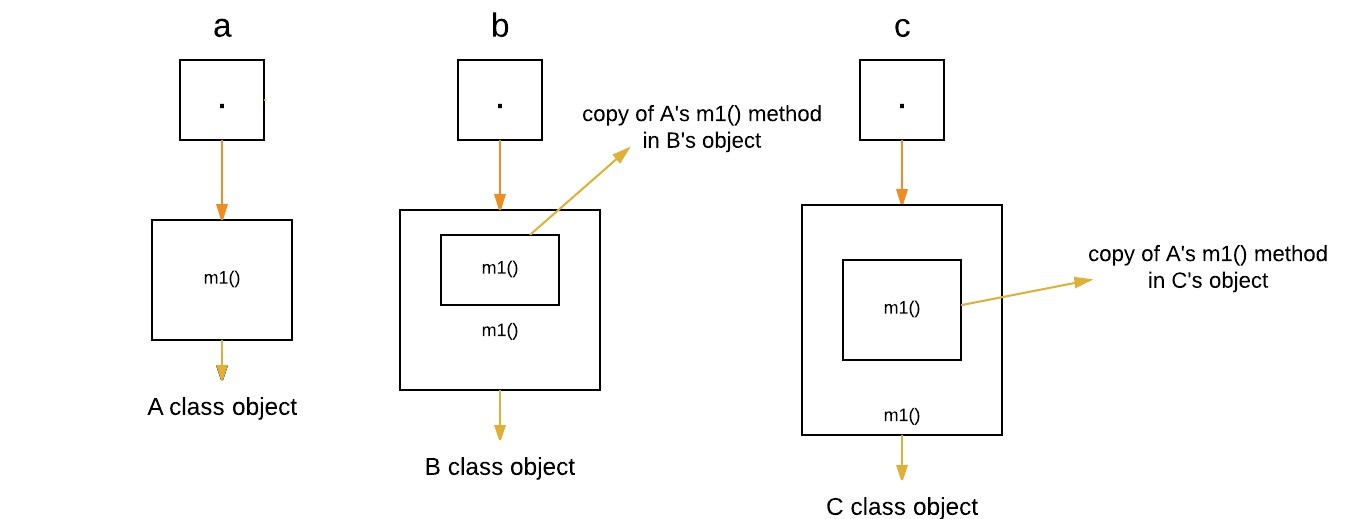
}

}

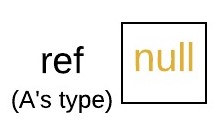
**Explanation :**

The above program creates one superclass called A and it’s two subclasses B and C. These subclasses overrides m1( ) method.

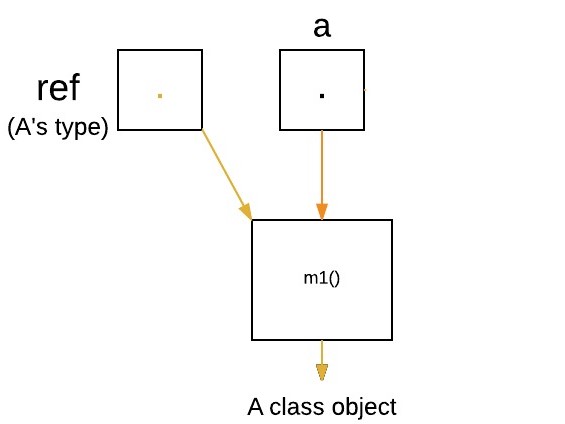
* 1. Inside the main() method in Dispatch class, initially objects of type A, B, and C are declared.
  2. A a = new A(); // object of type A
  3. B b = new B(); // object of type B
  4. C c = new C(); // object of type C



* 1. Now a reference of type A, called ref, is also declared, initially it will point to null.
  2. A ref; // obtain a reference of type A

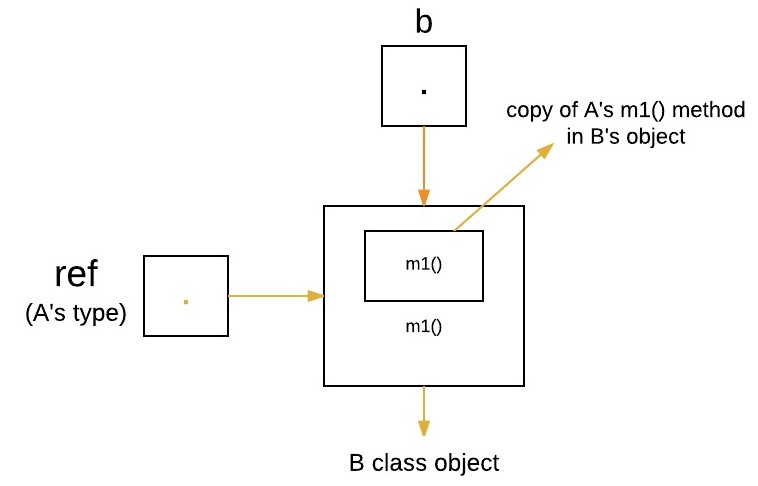
[](https://media.geeksforgeeks.org/wp-content/uploads/w.jpeg)

* 1. Now we are assigning a reference to each **type of object** (either A’s or B’s or C’s) to *ref*, one-by-one, and uses that reference to invoke m1( ). As the output shows, the version of m1( ) executed is determined **by the type of object being referred to at the time of the call.**
  2. ref = a; // r refers to an A object
  3. ref.m1(); // calling A's version of m1()

[](https://media.geeksforgeeks.org/wp-content/uploads/q1.jpeg)

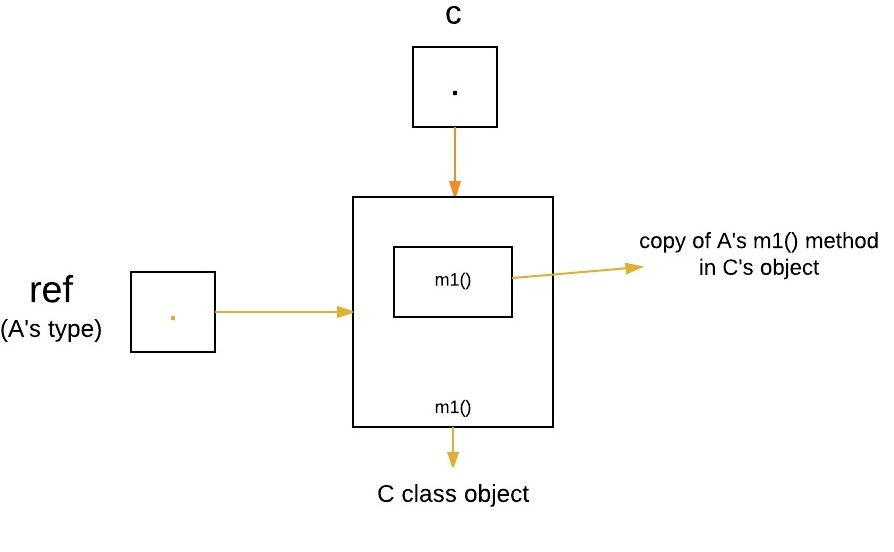
ref = b; // now r refers to a B object

ref.m1(); // calling B's version of m1()

[](https://media.geeksforgeeks.org/wp-content/uploads/q2.jpeg)

ref = c; // now r refers to a C object

ref.m1(); // calling C's version of m1()



**Runtime Polymorphism with Data Members**

In Java, we can override methods only, not the variables(data members), so **runtime polymorphism cannot be achieved by data members.** For example :

//Java program to illustrate the fact that

//runtime polymorphism cannot be achieved

//by data members

//class A

**class** A {

**int** x = 10;

}

//class B

**class** B **extends** A {

**int** x = 20;

}

//Driver class

**public** **class** Main {

**public** **static** **void** main(String args[]) {

A a = **new** B(); // object of type B

// Data member of class A will be accessed

System.***out***.println(a.x);

}

}

**Explanation :** In above program, both the class A(super class) and B(sub class) have a common variable ‘x’. Now we make object of class B, referred by ‘a’ which is of type of class A. Since variables are not overridden, so the statement “a.x” will **always** refer to data member of super class.

**Advantages of Dynamic Method Dispatch**

1. Dynamic method dispatch allow Java to support [overriding of methods](https://www.geeksforgeeks.org/overriding-in-java/) which is central for run-time polymorphism.
2. It allows a class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
3. It also allow subclasses to add its specific methods subclasses to define the specific implementation of some.

[Static vs Dynamic binding](https://www.geeksforgeeks.org/static-vs-dynamic-binding-in-java/)

* Static binding is done during compile-time while dynamic binding is done during run-time.
* private, final and static methods and variables uses static binding and bonded by compiler while overridden methods are bonded during runtime based upon type of runtime object

**Overriding equals method in Java**

**class** Complex {

**private** **double** re, im;

**public** Complex(**double** re, **double** im) {

**this**.re = re;

**this**.im = im;

}

}

// Driver class to test the Complex class

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Complex c1 = **new** Complex(10, 15);

Complex c2 = **new** Complex(10, 15);

**if** (c1 == c2) {

System.***out***.println("Equal ");

} **else** {

System.***out***.println("Not Equal ");

}

}

}

Output:

Not Equal

The reason for printing “Not Equal” is simple: when we compare c1 and c2, it is checked whether both c1 and c2 refer to same object or not ([object variables are always references in Java](https://www.geeksforgeeks.org/g-fact-46/)). c1 and c2 refer to two different objects, hence the value (c1 == c2) is false. If we create another reference say c3 like following, then (c1 == c3) will give true.

Complex c3 = c1;  // (c3 == c1) will be true

So, how do we check for equality of values inside the objects? All classes in Java inherit from the Object class, directly or indirectly (See point 1 of [this](https://www.geeksforgeeks.org/comparison-of-inheritance-in-c-and-java/)). The [Object class](http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Object.html) has some basic methods like clone(), toString(), equals(),.. etc. We can override the equals method in our class to check whether two objects have same data or not.

**class** Complex {

**private** **double** re, im;

**public** Complex(**double** re, **double** im) {

**this**.re = re;

**this**.im = im;

}

// Overriding equals() to compare two Complex objects

@Override

**public** **boolean** equals(Object o) {

// If the object is compared with itself then return true

**if** (o == **this**) {

**return** **true**;

}

/\*

\* Check if o is an instance of Complex or not "null instanceof [type]" also

\* returns false

\*/

**if** (!(o **instanceof** Complex)) {

**return** **false**;

}

// typecast o to Complex so that we can compare data members

Complex c = (Complex) o;

// Compare the data members and return accordingly

**return** Double.*compare*(re, c.re) == 0 && Double.*compare*(im, c.im) == 0;

}

}

// Driver class to test the Complex class

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Complex c1 = **new** Complex(10, 15);

Complex c2 = **new** Complex(10, 15);

**if** (c1.equals(c2)) {

System.***out***.println("Equal ");

} **else** {

System.***out***.println("Not Equal ");

}

}

}

Equal

As a side note, when we override equals(), it is recommended to also override the hashCode() method. If we don’t do so, equal objects may get different hash-values; and hash based collections, including HashMap, HashSet, and Hashtable do not work properly (see [this](http://www.technofundo.com/tech/java/equalhash.html)for more details). We will be coverig more about hashCode() in a separate post.

**Overriding toString() in Java**

This post is similar to [Overriding equals method in Java](https://www.geeksforgeeks.org/overriding-equals-method-in-java/). Consider the following Java program:

**class** Complex {

**private** **double** re, im;

**public** Complex(**double** re, **double** im) {

**this**.re = re;

**this**.im = im;

}

}

//Driver class to test the Complex class

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Complex c1 = **new** Complex(10, 15);

System.***out***.println(c1);

}

}

The output is, class name, then ‘at’ sign, and at the end [*hashCode*](http://en.wikipedia.org/wiki/Java_hashCode()) of object. All classes in Java inherit from the Object class, directly or indirectly (See point 1 of [this](https://www.geeksforgeeks.org/comparison-of-inheritance-in-c-and-java/)). The Object class has some basic methods like clone(), toString(), equals(),.. etc. The default toString() method in Object prints “class name @ hash code”. We can override toString() method in our class to print proper output. For example, in the following code toString() is overridden to print “Real + i Imag” form.

**class** Complex {

**private** **double** re, im;

**public** Complex(**double** re, **double** im) {

**this**.re = re;

**this**.im = im;

}

/\*

\* Returns the string representation of this Complex number. The format of

\* string is "Re + iIm" where Re is real part and Im is imagenary part.

\*/

@Override

**public** String toString() {

**return** String.*format*(re + " + i" + im);

}

}

//Driver class to test the Complex class

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Complex c1 = **new** Complex(10, 15);

System.***out***.println(c1);

}

}

In general, it is a good idea to override toString() as we get get proper output when an object is used in print() or println().

**Output of Java program | Set 18 (Overriding)**

**1) What is the output of the following program?**

**class** Derived {

**protected** **final** **void** getDetails() {

System.***out***.println("Derived class");

}

}

**public** **class** Main **extends** Derived {

**protected** **final** **void** getDetails() {

System.***out***.println("Test class");

}

**public** **static** **void** main(String[] args) {

Derived obj = **new** Derived();

obj.getDetails();

}

}

Ans : Compilation error

**Explanation:**Final and static methods cannot be overridden

**2) What is the output of the following program?**

**class** Derived {

**public** **void** getDetails(String temp) {

System.***out***.println("Derived class " + temp);

}

}

**public** **class** Main **extends** Derived {

**public** **int** getDetails(String temp) {

System.***out***.println("Test class " + temp);

**return** 0;

}

**public** **static** **void** main(String[] args) {

Main obj = **new** Main();

obj.getDetails("GFG");

}

}

Compilation error **:Explanation:**The overriding method must have same signature, which includes, the argument list and the return type.

**3) What is the output of the following program?**

**class** Derived {

**public** **void** getDetails() {

System.***out***.println("Derived class");

}

}

**public** **class** Test **extends** Derived {

**protected** **void** getDetails() {

System.***out***.println("Test class");

}

**public** **static** **void** main(String[] args) {

Derived obj = **new** Test(); // line xyz

obj.getDetails();

}

}

Compilation error due to access modifier

**Explanation:**The overriding method can not have more restrictive access modifier.

**4) What is the output of the following program?**

**class** Derived {

**public** **void** getDetails() **throws** IOException // line 23

{

System.***out***.println("Derived class");

}

}

**public** **class** Main **extends** Derived {

**public** **void** getDetails() **throws** Exception // line 24

{

System.***out***.println("Test class");

}

**public** **static** **void** main(String[] args) **throws** IOException // line 25

{

Derived obj = **new** Main();

obj.getDetails();

}

}

a) Compilation error due to line 23  
b) Compilation error due to line 24  
c) Compilation error due to line 25  
d) All the above

**Ans.**(b)  
**Explanation:**The exception thrown by the overriding method should not be new or more broader checked exception. In the code above, [Exception](https://www.geeksforgeeks.org/exceptions-in-java/) is more broader class of checked exception than IOException, so this results in compilation error.

**5) What is the output of the following program?**

**class** Derived {

**public** **void** getDetails() {

System.***out***.printf("Derived class ");

}

}

**public** **class** Main **extends** Derived {

**public** **void** getDetails() {

System.***out***.printf("Test class ");

**super**.getDetails();

}

**public** **static** **void** main(String[] args) {

Derived obj = **new** Main();

obj.getDetails();

}

}

a) Test class Derived class  
b) Derived class Test class  
c) Compilation error  
d) Runtime error

**Ans.**(a)  
**Explanation:**[super keyword](https://www.geeksforgeeks.org/super-keyword/)is used to invoke the overridden method from a child class explicitly.