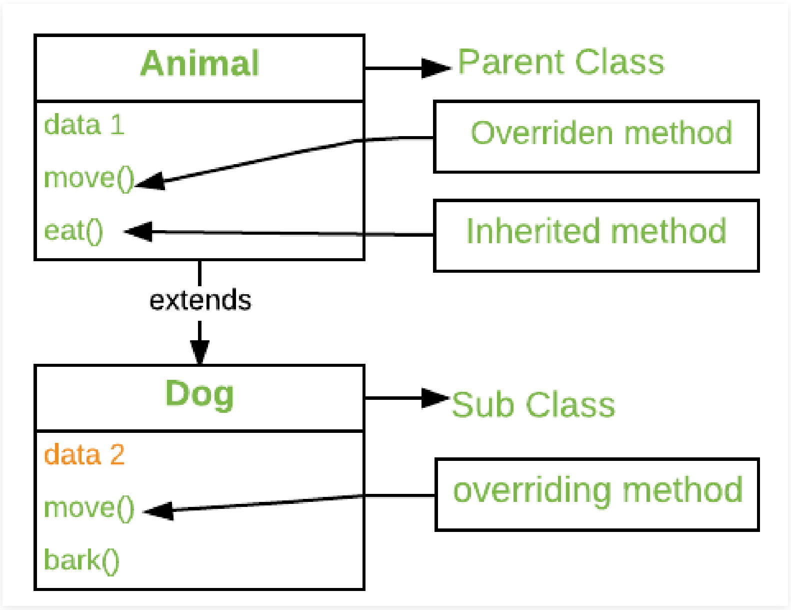
Overriding in Java

Difficulty Level : Medium Last Updated : 28 Jun, 2021

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.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
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();
    }
}
```

Output:

```
Parent's show()
Child's show()
```

Rules for method overriding:

1. **Overriding and Access-Modifiers :** The access modifier 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.

```
// A Simple Java program to demonstrate
// Overriding and Access-Modifiers
```

```
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
class Main {
    public static void main(String[] args)
    {
        Parent obj1 = new Parent();
        obj1.m2();
        Parent obj2 = new Child();
        obj2.m2();
    }
}
```

Output:

```
From parent m2()
From child m2()
```

2. **Final methods can not be overridden :** If we don't want a method to be overridden, we declare it as final. Please see Using final with Inheritance .

```
// A Java program to demonstrate that
// final methods cannot be overridden

class Parent {
    // Can't be overridden
    final void show() {}
}

class Child extends Parent {
    // This would produce error
    void show() {}
}
```

Output:

```
13: error: show() in Child cannot override show() in Parent
    void show() {  }
         ^
  overridden method is final
```

3. **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.
The following table summarizes what happens when you define a method with the same signature as a method in a super-class.

	Superclass Instance Method	Superclass Static Method
Subclass Instance Method	Overrides	Generates a compile-time error
Subclass Static Method	Generates a compile-time error	Hides

```
// 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();
    }
}
```

Output:

```
From parent static m1()
From child non-static(instance) m2()
```

4. **Private methods can not be overridden :** Private methods cannot be overridden as they are bonded during compile time. Therefore we can't even override private methods in a subclass. (See this for details).

5. **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**.

6. **Invoking overridden method from sub-class :** We can call parent class method in overriding method using super keyword.

```java
// 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();
    }
}
```

Output:

```
Parent's show()
Child's show()
```

7. **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).
8. **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, throwing checked exception will lead to compile-time error.

```
/* Java program to demonstrate overriding when
  superclass method does not declare an exception
*/

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
    // issue while throwin checked exception
    void m2() throws Exception
    {
        System.out.println("From child m2");
    }
}
```

Output:

```
error: m2() in Child cannot override m2() in Parent
    void m2() throws Exception{ System.out.println("From child m2");}
         ^
  overridden method does not throw Exception
```

- **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 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
    // issue while throwing parent exception
    void m1() throws Exception
    {
        System.out.println("From child4 m1()");
    }
}
```

Output:

```
error: m1() in Child4 cannot override m1() in Parent
    void m1() throws Exception
         ^
  overridden method does not throw Exception
```

9. **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.

10. **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 to achieve overriding or Run Time Polymorphism. In Java, methods are virtual by default.
2. We can have multilevel method-overriding.

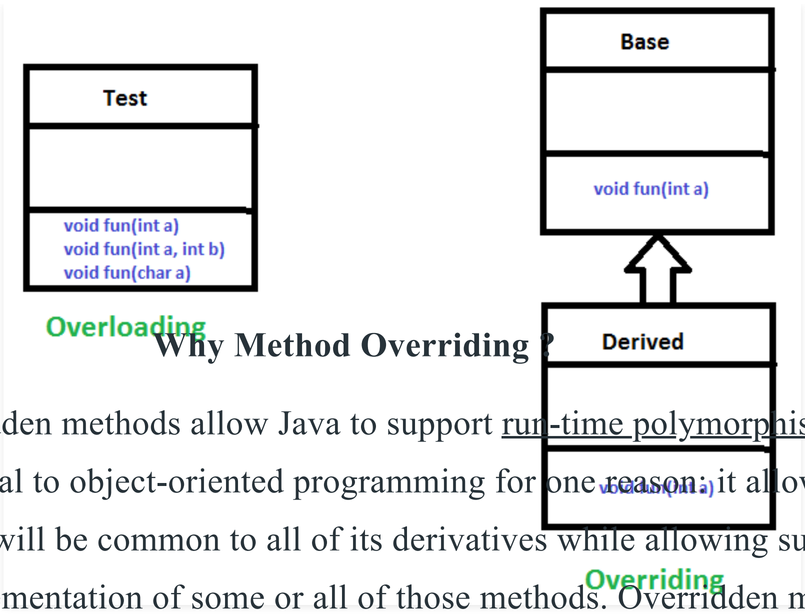
```
// A Java program to demonstrate
// multi-level overriding

// 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();
    }
}
```

Output:

```
GrandChild's show()
```

3. **Overriding vs** Overloading **:**

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



2. Overloading is an example of compiler-time polymorphism and overriding is an example of run time polymorphism.

Why Method Overriding ?

As stated earlier, overridden methods allow Java to support run-time polymorphism. 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.

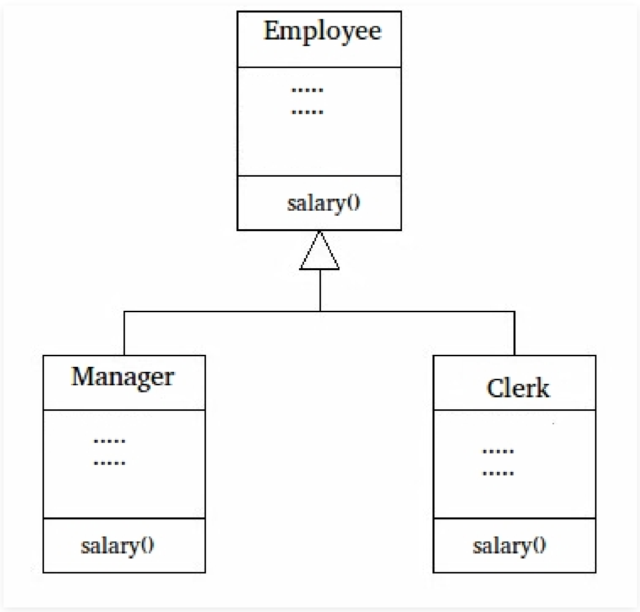
Dynamic Method Dispatch is one of the most powerful mechanisms that object-oriented design brings to bear on code reuse and robustness. The ability to exist code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful 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 : 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.



```java
// 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);
    }
}
```

Output:

```
Manager's salary : 30000
Clerk's salary : 20000
```

Related Article:

- Dynamic Method Dispatch or Runtime Polymorphism in Java
- Overriding equals() method of Object class
- Overriding toString() method of Object class
- Overloading in java
- Output of Java program | Set 18 (Overriding)

This article is contributed by **Twinkle Tyagi and Gaurav Miglani**. If you like GeeksforGeeks and would like to contribute, you can also write an article using write.geeksforgeeks.org or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.