**Inner class or nested class or non Static Inner class.**

It is possible to define a class within another class; such classes are known as *nested classes*. The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B does not exist independently of A. A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class.

**A nested class that is declared directly within its enclosing class scope is a member of its enclosing class. It is also possible to declare a nested class that is local to a block.**

There are two types of nested classes: ***static* and *non-static*.**

A static nested class is one that has the **static** modifier applied. Because it is static, it must access the non-static members of its enclosing class through an object. That is, it cannot refer to non-static members of its enclosing class directly. Because of this restriction, static nested classes are rarely used.

The most important type of nested class is the *inner* class. An inner class is a **non-static nested class.** It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

**What is inner class? Whether it can access all members of the outer class?**

**Java inner class** or **nested class** is a class that is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place to be more readable and maintainable.

Additionally, it can access all the members of the outer class, including private data members and methods.

Can outer class access inner class members ?

NO. outer class cannot access inner class members.

Syntax of Inner class or nested class

**class** Java\_Outer\_class

{   //code

**class** Java\_Inner\_class

{   //code

  }

}

### **What are Advantage of Java inner classes?**

There are three advantages of inner classes in Java. They are as follows:

1. Nested classes represent a particular type of relationship that is **it can access all the members (data members and methods) of the outer class,** including private.
2. Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.
3. **Code Optimization**: It requires less code to write.

**What is the Need of Java Inner class?**

Sometimes users need to program a class in such a way so that no other class can access it. Therefore, it would be better if you include it within other classes.

If all the class objects are a part of the outer object then it is easier to nest that class inside the outer class. That way all the outer class can access all the objects of the inner class.

// Demonstrate an inner class.

class Outer

{ int outer\_x = 100;

void test()

{ Inner inner = new Inner();

inner.display();

}

// this is an inner class

class Inner

{ void display()

{ System.out.println("display: outer\_x = " + outer\_x);

}

}

}

class InnerClassDemo

{ public static void main(String args[])

{ Outer outer = new Outer();

outer.test();

}

}

**Output from this application is shown here:**

display: outer\_x = 100

**Inheritance**

**A Superclass Variable Can Reference a Subclass Object**

A reference variable of a superclass can be assigned a reference to any subclass derived

from that superclass.

class RefDemo

{ public static void main(String args[])

{

BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);

Box plainbox = new Box();

double vol;

vol = weightbox.volume();

System.out.println("Volume of weightbox is " + vol);

System.out.println("Weight of weightbox is " +

weightbox.weight);

System.out.println();

// assign BoxWeight reference to Box reference

plainbox = weightbox;

vol = plainbox.volume(); // OK, volume() defined in Box

System.out.println("Volume of plainbox is " + vol);

/\* The following statement is invalid because plainbox

does not define a weight member. \*/

// System.out.println("Weight of plainbox is " + plainbox.weight);

}

}

Here, **weightbox** is a reference to **BoxWeight** objects, and **plainbox** is a reference to **Box**

objects. Since **BoxWeight** is a subclass of **Box**, it is permissible to assign **plainbox** a reference

to the **weightbox** object.

It is important to understand that it is the type of the reference variable—not the type

of the object that it refers to—that determines what members can be accessed. That is,

when a reference to a subclass object is assigned to a superclass reference variable, you will

have access only to those parts of the object defined by the superclass. This is why **plainbox**

can’t access **weight** even when it refers to a **BoxWeight** object. If you think about it, this

makes sense, because the superclass has no knowledge of what a subclass adds to it. This is

why the last line of code in the preceding fragment is commented out. It is not possible for

a **Box** reference to access the **weight** field, because **Box** does not define one.

**Using super**

Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**.

**super** has two general forms.

1. The first calls the superclass’ constructor.
2. The second is used to access a member of the superclass that has been hidden by a member of a subclass.

**Using super to Call Superclass Constructors**

A subclass can call a constructor defined by its superclass by use of the following form of

**super**:

super(*arg-list*);

Here, *arg-list* specifies any arguments needed by the constructor in the superclass.

**super( )** must always be the first statement executed inside a subclass’ constructor.

To see how **super( )** is used, consider this improved version of the **BoxWeight** class:

// BoxWeight now uses super to initialize its Box attributes.

class BoxWeight extends Box

{

double weight; // weight of box

// initialize width, height, and depth using super()

BoxWeight(double w, double h, double d, double m)

{ super(w, h, d); // call superclass constructor

weight = m;

}

}

// Complete program

// in each sub class constructor, super( ) is called using the appropriate arguments.

//Notice that width, height, and depth have been made private within Box.

// A complete implementation of BoxWeight.

class Box {

private double width;

private double height;

private double depth;

// construct clone of an object

Box(Box ob) { // pass object to constructor

width = ob.width;

height = ob.height;

depth = ob.depth;

}

// constructor used when all dimensions specified

Box(double w, double h, double d) {

width = w;

height = h;

depth = d;

}

// constructor used when no dimensions specified

Box() {

width = -1; // use -1 to indicate

height = -1; // an uninitialized

depth = -1; // box

}

// constructor used when cube is created

Box(double len) {

width = height = depth = len;

}

Box(Box ob) {

Width = ob.width; // use -1 to indicate

height = ob.height; // an uninitialized

depth = ob.depth; // box

}

// compute and return volume

double volume() {

return width \* height \* depth;

}

}

// BoxWeight now fully implements all constructors.

class BoxWeight extends Box {

double weight; // weight of box

// construct clone of an object

BoxWeight(BoxWeight ob) { // pass object to constructor

super(ob);

weight = ob.weight;

}

// constructor when all parameters are specified

BoxWeight(double w, double h, double d, double m) {

super(w, h, d); // call superclass constructor

weight = m;

}

// default constructor

BoxWeight() {

super();

weight = -1;

}

// constructor used when cube is created

BoxWeight(double len, double m) {

super(len);

weight = m;

}

}

public class DemoSuper {

public static void main(String args[]) {

BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);

BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);

BoxWeight mybox3 = new BoxWeight(); // default

BoxWeight mycube = new BoxWeight(3, 2);

BoxWeight myclone = new BoxWeight(mybox1);

double vol;

vol = mybox1.volume();

System.out.println("Volume of mybox1 is " + vol);

System.out.println("Weight of mybox1 is " + mybox1.weight);

System.out.println();

vol = mybox2.volume();

System.out.println("Volume of mybox2 is " + vol);

System.out.println("Weight of mybox2 is " + mybox2.weight);

System.out.println();

vol = mybox3.volume();

System.out.println("Volume of mybox3 is " + vol);

System.out.println("Weight of mybox3 is " + mybox3.weight);

System.out.println();

vol = myclone.volume();

System.out.println("Volume of myclone is " + vol);

System.out.println("Weight of myclone is " + myclone.weight);

System.out.println();

vol = mycube.volume();

System.out.println("Volume of mycube is " + vol);

System.out.println("Weight of mycube is " + mycube.weight);

System.out.println();

}

}

**Box(Box ob)**. As mentioned earlier, a superclass variable can be used to reference any object derived from that class.

Thus, we are able to pass a **BoxWeight** object to the **Box** constructor. Of course, **Box** only has knowledge of its own members. Let’s review the key concepts behind **super( )**. When a subclass calls **super( )**, it is calling the constructor of its immediate superclass. Thus, **super( )** always refers to the superclass immediately above the calling class. This is true even in a multileveled hierarchy. Also, **super( )** must always be the first statement executed inside a subclass constructor.

The second form of **super** acts somewhat like **this**, except that it always refers to the

superclass of the subclass in which it is used. This usage has the following general form:

super.*member*

Here, *member* can be either a method or an instance variable.

This second form of **super** is most applicable to situations in which member names

of a subclass hide members by the same name in the superclass. Consider this simple class

hierarchy:

// Using super to overcome name hiding.

class A

{

int i;

}

// Create a subclass by extending class A.\

class B extends A

{

int i; // this i hides the i in A

B(int a, int b)

{

super.i = a; // i in A

i = b; // i in B

}

void show()

{

System.out.println("i in superclass: " + super.i);

System.out.println("i in subclass: " + i);

}

}

class UseSuper

{

public static void main(String args[])

{

B subOb = new B(1, 2);

subOb.show();

}

}

This program displays the following:

i in superclass: 1

i in subclass: 2

Although the instance variable **i** in **B** hides the **i** in **A**, **super** allows access to the **i** defined in the superclass. As you will see, **super** can also be used to call methods that are hidden by a subclass.

**Multilevel Hierarchy**

**When Constructors Are Executed**

**Method Overriding**

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass. When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

Method overriding occurs *only* when the names and the type signatures of the two methods are identical.

If they are not, then the two methods are simply overloaded.

**Dynamic Method Dispatch**

Method overriding forms the basis for one of Java’s most powerful concepts: *dynamic method dispatch*.

***Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.***

**Dynamic method dispatch is important because this is how Java implements run-time polymorphism.**

Let’s begin by restating an important principle: a superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time.

Here is how. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called.

**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.**

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.

// Dynamic Method Dispatch

class A

{

void callme()

{

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

}

}

class B extends A

{

// override callme()

void callme()

{

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

}

}

class C extends A

{

// override callme()

void callme()

{

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

}

}

class Dispatch

{

public static void main(String args[])

{

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

B b = new B(); // object of type B

C c = new C(); // object of type C

A r; // obtain a reference of type A

r = a; // r refers to an A object

r.callme(); // calls A's version of callme

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

r.callme(); // calls B's version of callme

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

r.callme(); // calls C's version of callme

}

}

The output from the program is shown here:

Inside A's callme method

Inside B's callme method

Inside C's callme method

**Why Overridden Methods?**

**Using Abstract Classes**

sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement.

You can require that certain methods be overridden by subclasses by specifying the **abstract** type modifier. These methods are sometimes referred to as *subclasser responsibility* because they have **no implementation** specified in the superclass. Thus, a subclass must override them—it cannot simply use the version defined in the superclass. To declare an abstract method, use this general form:

***abstract type name(parameter-list);***

As you can see, no method body is present.

Any class that contains one or more abstract methods must also be declared abstract.

To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration. There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the **new** operator. Such objects would be useless, because an abstract class is not fully defined. Also, you cannot declare abstract constructors, or abstract static methods.

Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be declared **abstract** itself.

// A Simple demonstration of abstract.

abstract class A

{

abstract void callme();

// concrete methods are still allowed in abstract classes

void callmetoo()

{ System.out.println("This is a concrete method.");

}

}

class B extends A

{ void callme()

{ System.out.println("B's implementation of callme.");

}

}

class AbstractDemo

{ public static void main(String args[])

{

B b = new B();

b.callme();

b.callmetoo();

}

}

**Using final with Inheritance**

The keyword **final** has three uses. First, it can be used to create the equivalent of a named constant. This use was described in the preceding chapter. The other two uses of **final** apply to inheritance. Both are examined here.

**Using final to Prevent Overriding**

While method overriding is one of Java’s most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden. The following fragment illustrates **final**:

class A {

final void meth() {

System.out.println("This is a final method.");

}

}

class B extends A {

void meth() { // ERROR! Can't override.

System.out.println("Illegal!");

}

}

**Using final to Prevent Inheritance**

Sometimes you will want to prevent a class from being inherited. To do this, precede the

class declaration with **final**. Declaring a class as **final** implicitly declares all of its methods as

**final**, too. As you might expect, it is illegal to declare a class as both **abstract** and **final** since

an abstract class is incomplete by itself and relies upon its subclasses to provide complete

implementations.

Here is an example of a **final** class:

final class A {

//...

}

// The following class is illegal.

class B extends A { // ERROR! Can't subclass A

//...

}

As the comments imply, it is illegal for **B** to inherit **A** since **A** is declared as **final**.

**The Object Class**

There is one special class, **Object**, defined by Java. All other classes are subclasses of **Object**.

That is, **Object** is a superclass of all other classes. This means that a reference variable of

type **Object** can refer to an object of any other class. Also, since arrays are implemented as

classes, a variable of type **Object** can also refer to any array.

**Object** defines the following methods, which means that they are available in every object

**Method Purpose**

* Object clone( ) 🡪Creates a new object that is the same as the object being cloned.
* boolean equals(Object *object*) 🡪 Determines whether one object is equal to another.
* void finalize( ) 🡪Called before an unused object is recycled.
* Class<?> getClass( ) 🡪Obtains the class of an object at run time.
* int hashCode( ) 🡪Returns the hash code associated with the invoking object.
* void notify( ) 🡪Resumes execution of a thread waiting on the invoking object.
* void notifyAll( ) 🡪Resumes execution of all threads waiting on the invoking object.
* **String toString( )** 🡪**Returns a string that describes the object.**
* void wait( )
* void wait(long *milliseconds*)
* void wait(long *milliseconds*,
* int *nanoseconds*) 🡪 Waits on another thread of execution.

two methods now: **equals( )** and **toString( )**. The **equals( )** method compares two objects. It returns **true** if the objects are equal, and **false** otherwise. The precise definition of equality can vary, depending on the type of objects being compared. The **toString( )** method returns a string that contains a description of the object on which it is called. Also, this method is automatically called when an object is output using **println( )**. Many classes override this method. Doing so allows them to tailor a description specifically for the types of objects that they create.

# Java Abstract Class and Abstract Methods

## Java Abstract Class

The abstract class in Java cannot be instantiated (we cannot create objects of abstract classes). We use the abstract keyword to declare an abstract class. For example,

// create an abstract class

abstract class Language {

// fields and methods

}

...

// try to create an object Language

// throws an error

Language obj = new Language();

An abstract class can have both the regular methods and abstract methods. For example,

abstract class Language {

// abstract method

abstract void method1();

// regular method

void method2() {

System.out.println("This is regular method");

}

}

To know about the non-abstract methods, visit [Java methods](https://www.programiz.com/java-programming/methods). Here, we will learn about abstract methods.

## Java Abstract Method

A method that doesn't have its body is known as an abstract method. We use the same abstract keyword to create abstract methods. For example,

abstract void display();

Here, display() is an abstract method. The body of display() is replaced by ;.

If a class contains an abstract method, then the class should be declared abstract. Otherwise, it will generate an error. For example,

// error

// class should be abstract

class Language {

// abstract method

abstract void method1();

}

### Example: Java Abstract Class and Method

Though abstract classes cannot be instantiated, we can create subclasses from it. We can then access members of the abstract class using the object of the subclass. For example,

abstract class Language {

// method of abstract class

public void display() {

System.out.println("This is Java Programming");

}

}

class Main extends Language {

public static void main(String[] args) {

// create an object of Main

Main obj = new Main();

// access method of abstract class

// using object of Main class

obj.display();

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**

This is Java programming

In the above example, we have created an abstract class named Language. The class contains a regular method display().

We have created the Main class that inherits the abstract class. Notice the statement,

obj.display();

Here, obj is the object of the child class Main. We are calling the method of the abstract class using the object obj.

## Implementing Abstract Methods

If the abstract class includes any abstract method, then all the child classes inherited from the abstract superclass must provide the implementation of the abstract method. For example,

abstract class Animal {

abstract void makeSound();

public void eat() {

System.out.println("I can eat.");

}

}

class Dog extends Animal {

// provide implementation of abstract method

public void makeSound() {

System.out.println("Bark bark");

}

}

class Main {

public static void main(String[] args) {

// create an object of Dog class

Dog d1 = new Dog();

d1.makeSound();

d1.eat();

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**

Bark bark

I can eat.

In the above example, we have created an abstract class Animal. The class contains an abstract method makeSound() and a non-abstract method eat().

We have inherited a subclass Dog from the superclass Animal. Here, the subclass Dog provides the implementation for the abstract method makeSound().

We then used the object d1 of the Dog class to call methods makeSound() and eat().

**Note**: If the Dog class doesn't provide the implementation of the abstract method makeSound(), Dog should also be declared as abstract. This is because the subclass Dog inherits makeSound() from Animal.

### Accesses Constructor of Abstract Classes

An abstract class can have constructors like the regular class. And, we can access the constructor of an abstract class from the subclass using the super keyword. For example,

abstract class Animal {

Animal() {

….

}

}

class Dog extends Animal {

Dog() {

super();

...

}

}

Here, we have used the super() inside the constructor of Dog to access the constructor of the Animal.

Note that the super should always be the first statement of the subclass constructor. Visit [Java super keyword](https://www.programiz.com/java-programming/super-keyword) to learn more.

## Java Abstraction

The major use of abstract classes and methods is to achieve abstraction in Java.

Abstraction is an important concept of object-oriented programming that allows us to hide unnecessary details and only show the needed information.

This allows us to manage complexity by omitting or hiding details with a simpler, higher-level idea.

A practical example of abstraction can be motorbike brakes. We know what brake does. When we apply the brake, the motorbike will stop. However, the working of the brake is kept hidden from us.

The major advantage of hiding the working of the brake is that now the manufacturer can implement brake differently for different motorbikes, however, what brake does will be the same.

Let's take an example that helps us to better understand Java abstraction.

### Example 3: Java Abstraction

abstract class MotorBike {

abstract void brake();

}

class SportsBike extends MotorBike {

// implementation of abstract method

public void brake() {

System.out.println("SportsBike Brake");

}

}

class MountainBike extends MotorBike {

// implementation of abstract method

public void brake() {

System.out.println("MountainBike Brake");

}

}

class Main {

public static void main(String[] args) {

MountainBike m1 = new MountainBike();

m1.brake();

SportsBike s1 = new SportsBike();

s1.brake();

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**:

MountainBike Brake

SportsBike Brake

In the above example, we have created an abstract super class MotorBike. The superclass MotorBike has an abstract method brake().

The brake() method cannot be implemented inside MotorBike. It is because every bike has different implementation of brakes. So, all the subclasses of MotorBike would have different implementation of brake().

So, the implementation of brake() in MotorBike is kept hidden.

Here, MountainBike makes its own implementation of brake() and SportsBike makes its own implementation of brake().

**Note**: We can also use interfaces to achieve abstraction in Java. To learn more, visit [Java Interface](https://www.programiz.com/java-programming/interfaces).

## Key Points to Remember

* We use the abstract keyword to create abstract classes and methods.
* An abstract method doesn't have any implementation (method body).
* A class containing abstract methods should also be abstract.
* We cannot create objects of an abstract class.
* To implement features of an abstract class, we inherit subclasses from it and create objects of the subclass.
* A subclass must override all abstract methods of an abstract class. However, if the subclass is declared abstract, it's not mandatory to override abstract methods.
* We can access the static attributes and methods of an abstract class using the reference of the abstract class. For example,

Animal.staticMethod();

# **Java Interface**

An interface is a fully abstract class. It includes a group of abstract methods (methods without a body).

We use the interface keyword to create an interface in Java. For example,

interface Language {

public void getType();

public void getVersion();

}

Here,

* Language is an interface.
* It includes abstract methods: getType() and getVersion().

## Implementing an Interface

Like abstract classes, we cannot create objects of interfaces.

To use an interface, other classes must implement it. We use the **implements** keyword to implement an interface.

### Example 1: Java Interface

interface Polygon {

void getArea(int length, int breadth);

}

// implement the Polygon interface

class Rectangle implements Polygon {

// implementation of abstract method

public void getArea(int length, int breadth) {

System.out.println("The area of the rectangle is " + (length \* breadth));

}

}

class Main {

public static void main(String[] args) {

Rectangle r1 = new Rectangle();

r1.getArea(5, 6);

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**

The area of the rectangle is 30

In the above example, we have created an interface named Polygon. The interface contains an abstract method getArea().

Here, the Rectangle class implements Polygon. And, provides the implementation of the getArea() method.

### Example 2: Java Interface

// create an interface

interface Language {

void getName(String name);

}

// class implements interface

class ProgrammingLanguage implements Language {

// implementation of abstract method

public void getName(String name) {

System.out.println("Programming Language: " + name);

}

}

class Main {

public static void main(String[] args) {

ProgrammingLanguage language = new ProgrammingLanguage();

language.getName("Java");

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**

Programming Language: Java

In the above example, we have created an interface named Language. The interface includes an abstract method getName().

Here, the ProgrammingLanguage class implements the interface and provides the implementation for the method.

### Implementing Multiple Interfaces

In Java, a class can also implement multiple interfaces. For example,

interface A {

// members of A

}

interface B {

// members of B

}

class C implements A, B {

// abstract members of A

// abstract members of B

}

## Extending an Interface

Similar to classes, interfaces can extend other interfaces. The extends keyword is used for extending interfaces. For example,

interface Line {

// members of Line interface

}

// extending interface

interface Polygon extends Line {

// members of Polygon interface

// members of Line interface

}

Here, the Polygon interface extends the Line interface. Now, if any class implements Polygon, it should provide implementations for all the abstract methods of both Line and Polygon.

### Extending Multiple Interfaces

An interface can extend multiple interfaces. For example,

interface A {

...

}

interface B {

...

}

interface C extends A, B {

...

}

## Advantages of Interface in Java

Now that we know what interfaces are, let's learn about why interfaces are used in Java.

* Similar to abstract classes, interfaces help us to achieve **abstraction in Java**.  
  Here, we know getArea() calculates the area of polygons but the way area is calculated is different for different polygons. Hence, the implementation of getArea() is independent of one another.
* Interfaces **provide specifications** that a class (which implements it) must follow.  
  In our previous example, we have used getArea() as a specification inside the interface Polygon. This is like setting a rule that we should be able to get the area of every polygon.  
    
  Now any class that implements the Polygon interface must provide an implementation for the getArea() method.
* Interfaces are also used to achieve multiple inheritance in Java. For example,
* interface Line {
* …
* }
* interface Polygon {
* …
* }
* class Rectangle implements Line, Polygon {
* …

}

Here, the class Rectangle is implementing two different interfaces. This is how we achieve multiple inheritance in Java.

***Note: All the methods inside an interface are implicitly public and all fields are implicitly public static final. For example,***

interface Language {

// by default public static final

String type = "programming language";

// by default public

void getName();

}

## **default methods in Java Interfaces**

With the release of Java 8, we can now add methods with implementation inside an interface. These methods are called default methods.

To declare default methods inside interfaces, we use the ***default*** keyword. For example,

public default void getSides() {

// body of getSides()

} Why default methods?

Let's take a scenario to understand why default methods are introduced in Java.

Suppose, we need to add a new method in an interface.

We can add the method in our interface easily without implementation. However, that's not the end of the story. All our classes that implement that interface must provide an implementation for the method.

If a large number of classes were implementing this interface, we need to track all these classes and make changes to them. This is not only tedious but error-prone as well.

To resolve this, Java introduced default methods. Default methods are inherited like ordinary methods.

Let's take an example to have a better understanding of default methods.

### Example: Default Method in Java Interface

interface Polygon {

void getArea();

// default method

default void getSides() {

System.out.println("I can get sides of a polygon.");

}

}

// implements the interface

class Rectangle implements Polygon {

public void getArea() {

int length = 6;

int breadth = 5;

int area = length \* breadth;

System.out.println("The area of the rectangle is " + area);

}

// overrides the getSides()

public void getSides() {

System.out.println("I have 4 sides.");

}

}

// implements the interface

class Square implements Polygon {

public void getArea() {

int length = 5;

int area = length \* length;

System.out.println("The area of the square is " + area);

}

}

class DefaultMethodDemo {

public static void main(String[] args) {

// create an object of Rectangle

Rectangle r1 = new Rectangle();

r1.getArea();

r1.getSides();

// create an object of Square

Square s1 = new Square();

s1.getArea();

s1.getSides();

}

}

**Output**

The area of the rectangle is 30

I have 4 sides.

The area of the square is 25

I can get sides of a polygon.

In the above example, we have created an interface named Polygon. It has a default method getSides() and an abstract method getArea().

Here, we have created two classes Rectangle and Square that implement Polygon.

The Rectangle class provides the implementation of the getArea() method and overrides the getSides() method. However, the Square class only provides the implementation of the getArea() method.

Now, while calling the getSides() method using the Rectangle object, the overridden method is called. However, in the case of the Square object, the default method is called.

## **private and static Methods in Interface**

The Java 8 also added another feature to include static methods inside an interface.

Similar to a class, we can access static methods of an interface using its references. For example,

// create an interface

interface Polygon {

staticMethod(){..}

}

// access static method

Polygon.staticMethod();

**Note**: With the release of Java 9, private methods are also supported in interfaces.

We cannot create objects of an interface. Hence, private methods are used as helper methods that provide support to other methods in interfaces.

### Practical Example of Interface

Let's see a more practical example of Java Interface.

// To use the sqrt function

import java.lang.Math;

interface Polygon {

void getArea();

// calculate the perimeter of a Polygon

default void getPerimeter(int... sides) {

int perimeter = 0;

for (int side: sides) {

perimeter += side;

}

System.out.println("Perimeter: " + perimeter);

}

}

class Triangle implements Polygon {

private int a, b, c;

private double s, area;

// initializing sides of a triangle

Triangle(int a, int b, int c) {

this.a = a;

this.b = b;

this.c = c;

s = 0;

}

// calculate the area of a triangle

public void getArea() {

s = (double) (a + b + c)/2;

area = Math.sqrt(s\*(s-a)\*(s-b)\*(s-c));

System.out.println("Area: " + area);

}

}

class Main {

public static void main(String[] args) {

Triangle t1 = new Triangle(2, 3, 4);

// calls the method of the Triangle class

t1.getArea();

// calls the method of Polygon

t1.getPerimeter(2, 3, 4);

}

}

**Output**

Area: 2.9047375096555625

Perimeter: 9

In the above program, we have created an interface named Polygon. It includes a default method getPerimeter() and an abstract method getArea().

We can calculate the perimeter of all polygons in the same manner so we implemented the body of getPerimeter() in Polygon.

Now, all polygons that implement Polygon can use getPerimeter() to calculate perimeter.

However, the rule for calculating the area is different for different polygons. Hence, getArea() is included without implementation.

Any class that implements Polygon must provide an implementation of getArea().

# Java Polymorphism

Polymorphism is an important concept of object-oriented programming. It simply means more than one form.

That is, the same entity (method or operator or object) can perform different operations in different scenarios.

## Example: Java Polymorphism

class Polygon {

// method to render a shape

public void render() {

System.out.println("Rendering Polygon...");

}

}

class Square extends Polygon {

// renders Square

public void render() {

System.out.println("Rendering Square...");

}

}

class Circle extends Polygon {

// renders circle

public void render() {

System.out.println("Rendering Circle...");

}

}

class Main {

public static void main(String[] args) {

// create an object of Square

Square s1 = new Square();

s1.render();

// create an object of Circle

Circle c1 = new Circle();

c1.render();

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**

Rendering Square...

Rendering Circle...

In the above example, we have created a superclass: Polygon and two subclasses: Square and Circle. Notice the use of the render() method.

The main purpose of the render() method is to render the shape. However, the process of rendering a square is different than the process of rendering a circle.

Hence, the render() method behaves differently in different classes. Or, we can say render() is polymorphic.

### Why Polymorphism?

Polymorphism allows us to create consistent code. In the previous example, we can also create different methods: renderSquare() and renderCircle() to render Square and Circle, respectively.

This will work perfectly. However, for every shape, we need to create different methods. It will make our code inconsistent.

To solve this, polymorphism in Java allows us to create a single method render() that will behave differently for different shapes.

**Note**: The print() method is also an example of polymorphism. It is used to print values of different types like char, int, string, etc.

We can achieve polymorphism in Java using the following ways:

1. [Method Overriding](https://www.programiz.com/java-programming/method-overriding)
2. [Method Overloading](https://www.programiz.com/java-programming/method-overloading)
3. Operator Overloading

## **Java Method Overriding**

During [inheritance in Java](https://www.programiz.com/java-programming/inheritance), if the same method is present in both the superclass and the subclass. Then, the method in the subclass overrides the same method in the superclass. This is called method overriding.

In this case, the same method will perform one operation in the superclass and another operation in the subclass. For example,

### Example 1: Polymorphism using method overriding

class Language {

public void displayInfo() {

System.out.println("Common English Language");

}

}

class Java extends Language {

@Override

public void displayInfo() {

System.out.println("Java Programming Language");

}

}

class Main {

public static void main(String[] args) {

// create an object of Java class

Java j1 = new Java();

j1.displayInfo();

// create an object of Language class

Language l1 = new Language();

l1.displayInfo();

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**:

Java Programming Language

Common English Language

In the above example, we have created a superclass named Language and a subclass named Java. Here, the method displayInfo() is present in both Language and Java.

The use of displayInfo() is to print the information. However, it is printing different information in Language and Java.

Based on the object used to call the method, the corresponding information is printed.

Working of Java Polymorphism

**Note**: The method that is called is determined during the execution of the program. Hence, method overriding is a **run-time polymorphism**.

## **2. Java Method Overloading**

In a Java class, we can create methods with the same name if they differ in parameters. For example,

void func() { ... }

void func(int a) { ... }

float func(double a) { ... }

float func(int a, float b) { ... }

This is known as method overloading in Java. Here, the same method will perform different operations based on the parameter.

### Example 3: Polymorphism using method overloading

class Pattern {

// method without parameter

public void display() {

for (int i = 0; i < 10; i++) {

System.out.print("\*");

}

}

// method with single parameter

public void display(char symbol) {

for (int i = 0; i < 10; i++) {

System.out.print(symbol);

}

}

}

class Main {

public static void main(String[] args) {

Pattern d1 = new Pattern();

// call method without any argument

d1.display();

System.out.println("\n");

// call method with a single argument

d1.display('#');

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**:

\*\*\*\*\*\*\*\*\*\*

##########

In the above example, we have created a class named Pattern. The class contains a method named display() that is overloaded.

// method with no arguments

display() {...}

// method with a single char type argument

display(char symbol) {...}

Here, the main function of display() is to print the pattern. However, based on the arguments passed, the method is performing different operations:

* prints a pattern of \*, if no argument is passed or
* prints pattern of the parameter, if a single char type argument is passed.

**Note**: The method that is called is determined by the compiler. Hence, it is also known as compile-time polymorphism.

## **3. Java Operator Overloading**

Some operators in Java behave differently with different operands. For example,

* + operator is overloaded to perform numeric addition as well as string concatenation, and
* operators like &, |, and ! are overloaded for logical and bitwise operations.

Let's see how we can achieve polymorphism using operator overloading.

The + operator is used to add two entities. However, in Java, the + operator performs two operations.

1. When + is used with numbers (integers and floating-point numbers), it performs mathematical addition. For example,

int a = 5;

int b = 6;

// + with numbers

int sum = a + b; // Output = 11

2. When we use the + operator with strings, it will perform string concatenation (join two strings). For example,

String first = "Java ";

String second = "Programming";

// + with strings

name = first + second; // Output = Java Programming

Here, we can see that the + operator is overloaded in Java to perform two operations: **addition** and **concatenation**.

**Note**: In languages like C++, we can define operators to work differently for different operands. However, Java doesn't support user-defined operator overloading.

## **Polymorphic Variables**

A variable is called polymorphic if it refers to different values under different conditions.

Object variables (instance variables) represent the behavior of polymorphic variables in Java. It is because object variables of a class can refer to objects of its class as well as objects of its subclasses.

### Example: Polymorphic Variables

class ProgrammingLanguage {

public void display() {

System.out.println("I am Programming Language.");

}

}

class Java extends ProgrammingLanguage {

@Override

public void display() {

System.out.println("I am Object-Oriented Programming Language.");

}

}

class Main {

public static void main(String[] args) {

// declare an object variable

ProgrammingLanguage pl;

// create object of ProgrammingLanguage

pl = new ProgrammingLanguage();

pl.display();

// create object of Java class

pl = new Java();

pl.display();

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**:

I am Programming Language.

I am Object-Oriented Programming Language.

In the above example, we have created an object variable pl of the ProgrammingLanguage class. Here, pl is a polymorphic variable. This is because,

* In statement pl = new ProgrammingLanguage(), pl refer to the object of the ProgrammingLanguage class.
* And, in statement pl = new Java(), pl refer to the object of the Java class.

# Java Encapsulation

## Java Encapsulation

Encapsulation is one of the key features of object-oriented programming. Encapsulation refers to the bundling of fields and methods inside a single class.

It prevents outer classes from accessing and changing fields and methods of a class. This also helps to achieve **data hiding**.

### Example 1: Java Encapsulation

class Area {

// fields to calculate area

int length;

int breadth;

// constructor to initialize values

Area(int length, int breadth) {

this.length = length;

this.breadth = breadth;

}

// method to calculate area

public void getArea() {

int area = length \* breadth;

System.out.println("Area: " + area);

}

}

class Main {

public static void main(String[] args) {

// create object of Area

// pass value of length and breadth

Area rectangle = new Area(5, 6);

rectangle.getArea();

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**

Area: 30

In the above example, we have created a class named Area. The main purpose of this class is to calculate the area.

To calculate an area, we need two variables: length and breadth and a method: getArea(). Hence, we bundled these fields and methods inside a single class.

Here, the fields and methods can be accessed from other classes as well. Hence, this is not **data hiding**.

This is only **encapsulation**. We are just keeping similar codes together.

**Note**: People often consider encapsulation as data hiding, but that's not entirely true.

Encapsulation refers to the bundling of related fields and methods together. This can be used to achieve data hiding. Encapsulation in itself is not data hiding.

## Why Encapsulation?

* In Java, encapsulation helps us to keep related fields and methods together, which makes our code cleaner and easy to read.
* It helps to control the values of our data fields. For example,
* class Person {
* private int age;
* public void setAge(int age) {
* if (age >= 0) {
* this.age = age;
* }
* }

}

Here, we are making the age variable private and applying logic inside the setAge() method. Now, age cannot be negative.

* The getter and setter methods provide **read-only** or **write-only** access to our class fields. For example,
* getName() // provides read-only access

setName() // provides write-only access

* It helps to decouple components of a system. For example, we can encapsulate code into multiple bundles.  
    
  These decoupled components (bundle) can be developed, tested, and debugged independently and concurrently. And, any changes in a particular component do not have any effect on other components.
* We can also achieve data hiding using encapsulation. In the above example, if we change the length and breadth variable into private, then the access to these fields is restricted.  
    
  And, they are kept hidden from outer classes. This is called **data hiding**.

## Data Hiding

Data hiding is a way of restricting the access of our data members by hiding the implementation details. Encapsulation also provides a way for data hiding.

We can use [access modifiers](https://www.programiz.com/java-programming/access-modifiers) to achieve data hiding. For example,

### Example 2: Data hiding using the private specifier

class Person {

// private field

private int age;

// getter method

public int getAge() {

return age;

}

// setter method

public void setAge(int age) {

this.age = age;

}

}

class Main {

public static void main(String[] args) {

// create an object of Person

Person p1 = new Person();

// change age using setter

p1.setAge(24);

// access age using getter

System.out.println("My age is " + p1.getAge());

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**

My age is 24

In the above example, we have a ***private*** field***age***. Since it is private, it cannot be accessed from outside the class.

In order to access ***age***, we have used public methods: **getAge()** and **setAge()**. These methods are called getter and setter methods.

Making ***age*** private allowed us to restrict unauthorized access from outside the class. This is ***data hiding***.

If we try to access the ***age*** field from the Main class, we will get an error.

// error: age has private access in Person

p1.age = 24;

## **Java Packages & API**

A package in Java is used to group related classes. Think of it as **a folder in a file directory**. We use packages to avoid name conflicts, and to write a better maintainable code. Packages are divided into two categories:

* Built-in Packages (packages from the Java API)
* User-defined Packages (create your own packages)

## **Built-in Packages**

The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment.

The library contains components for managing input, database programming, and much more



* [java.applet](https://docs.oracle.com/javase/8/docs/api/java/applet/package-frame.html), [java.awt](https://docs.oracle.com/javase/8/docs/api/java/awt/package-frame.html), [java.awt.event](https://docs.oracle.com/javase/8/docs/api/java/awt/event/package-frame.html), [java.awt.font](https://docs.oracle.com/javase/8/docs/api/java/awt/font/package-frame.html), [java.awt.image](https://docs.oracle.com/javase/8/docs/api/java/awt/image/package-frame.html), [java.beans](https://docs.oracle.com/javase/8/docs/api/java/beans/package-frame.html), [java.io](https://docs.oracle.com/javase/8/docs/api/java/io/package-frame.html), [java.lang](https://docs.oracle.com/javase/8/docs/api/java/lang/package-frame.html), , [java.math](https://docs.oracle.com/javase/8/docs/api/java/math/package-frame.html), [java.net](https://docs.oracle.com/javase/8/docs/api/java/net/package-frame.html), [java.nio](https://docs.oracle.com/javase/8/docs/api/java/nio/package-frame.html), [java.rmi](https://docs.oracle.com/javase/8/docs/api/java/rmi/package-frame.html), [java.sql](https://docs.oracle.com/javase/8/docs/api/java/sql/package-frame.html), [java.text](https://docs.oracle.com/javase/8/docs/api/java/text/package-frame.html), [java.text.spi](https://docs.oracle.com/javase/8/docs/api/java/text/spi/package-frame.html), [java.time.zone](https://docs.oracle.com/javase/8/docs/api/java/time/zone/package-frame.html), [java.util](https://docs.oracle.com/javase/8/docs/api/java/util/package-frame.html), [java.util.concurrent](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-frame.html), [java.util.regex](https://docs.oracle.com/javase/8/docs/api/java/util/regex/package-frame.html),
* The library is divided into **packages** and **classes**. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.
* To use a class or a package from the library, you need to use the import keyword:

### **Syntax**

import package.name.Class; // Import a single class

import package.name.\*; // Import the whole package

## **Import a Class**

The specific class from a package be imported using package.classname;

Ex.

import java.util.Scanner;

In the example above, java.util is a package, while Scanner is a class of the java.util package.

## **Import a Package**

To import a whole package, end the sentence with an asterisk sign (\*).

import java.util.\*;

## **User-defined Packages**

To create your own package, you need to understand that Java uses a file system directory to store them. Just like folders on your computer:

### **Example**

└── root

└── mypack

└── MyPackageClass.java

To create a package, use the package keyword:

### **MyPackageClass.java**

package mypack;

public class MyPackageClass {

public static void main(String[] args) {

System.out.println("This is my package!");

}

}

Save the file as MyPackageClass.java, and compile it:

C:\Users\Java>javac MyPackageClass.java

Then compile the package:

Syntax: javac -d directory javafilename

Directory name. If you give . then within the same (current) directory.

C:\Users\Java>javac -d . MyPackageClass.java

This forces the compiler to create the "mypack" package.

The -d keyword specifies the destination for where to save the class file. You can use any directory name, like c:/user/, or, if you want to keep the package within the same directory, you can use the dot sign ".", like in the example above.

**Note:** The package name should be written in lower case to avoid conflict with class names.

When we compiled the package in the example above, a new folder was created, called "mypack".

To run the **MyPackageClass.java** file, write the following:

C:\Users\Java>java mypack.MyPackageClas

## **How to access package from another package?**

There are three ways to access the package from outside the package.

1. import package.\*;
2. import package.classname;
3. fully qualified name.

#### 1) Using packagename.\*

If you use package.\* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

## **Example of package that import the packagename.\***

//save by A.java

**package** pack;

**public** **class** A

{   **public** **void** msg()

{ System.out.println("Hello");

}

}

---------------------------------------------------------------

-------------------------------------------------------------

**import** mypack.\*;

**import** pack.\*;

**public class** B

{   **public** **static** **void** display()

{    A obj1 = **new** A();

    obj.msg();

B obj2 =new B();

obj2.display();

   }

}

