A **package** is a collection of related classes

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

Import    java.util.* ;
```

package.name

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

```
import java.util . Scanner;
```

package.name

Class Name

out is a public static instance / object of the PrintStream class.

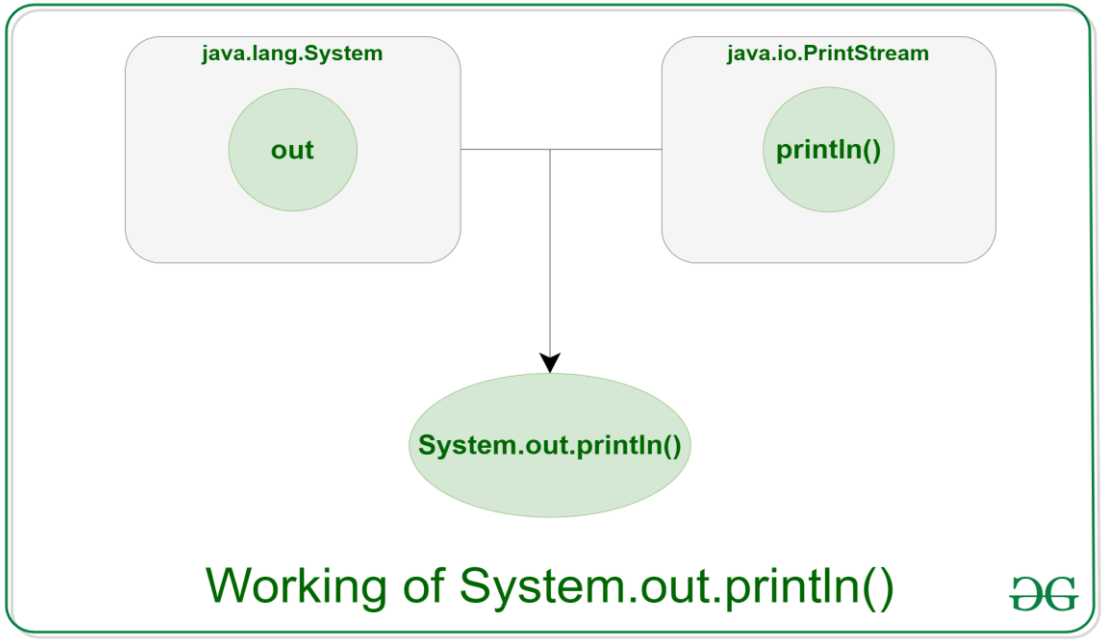
From PrintStream Class

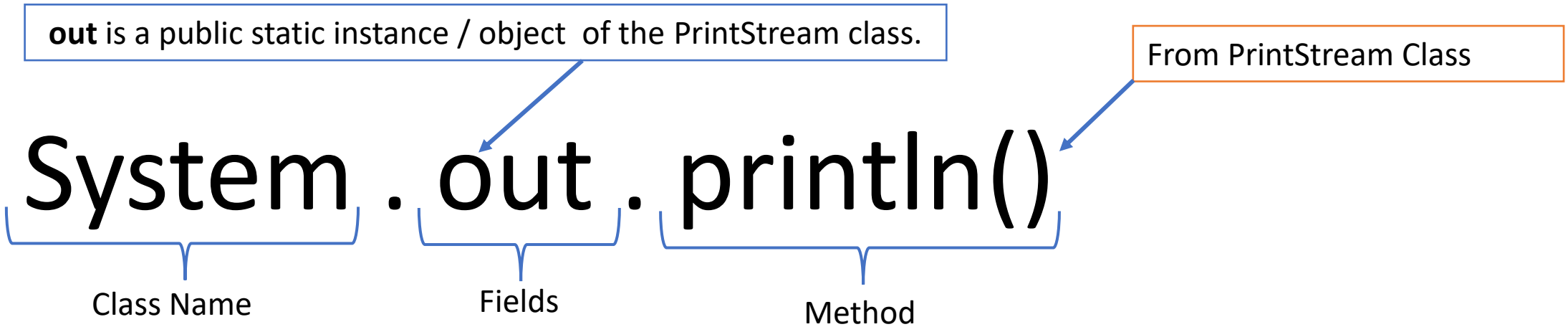
```
System . out . println()
```

Class Name — Fields — Method

Attributes / Class Variable / Field Member

All are Same in Java



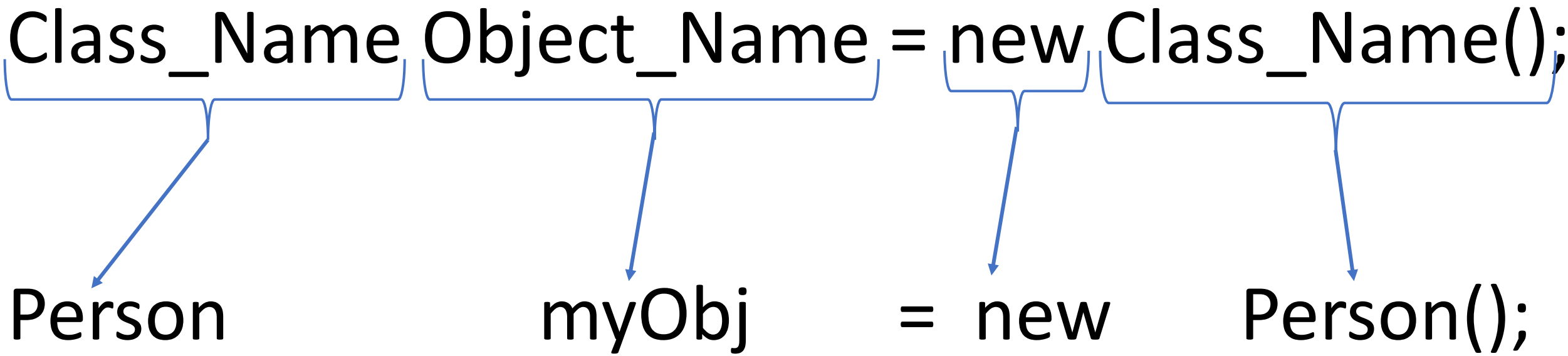


1.System: It is a final class defined in the java.lang package.

2.out: This is an instance of PrintStream type, which is a public and static member field of the System class.

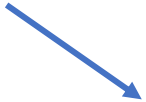
3.println(): As all instances of PrintStream class have a public method println(), hence we can invoke the same on out as well. This is an upgraded version of print(). It prints any argument passed to it and adds a new line to the output. We can assume that System.out represents the Standard Output Stream.

```
Class_Name Object_Name = new Class_Name();
```



Accessing Field and Method Using Object

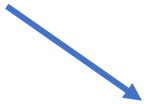
Object_Name . Field_Name = “Value";



To access Field

myObj.name = "John";

Object_Name . Method_Name ();

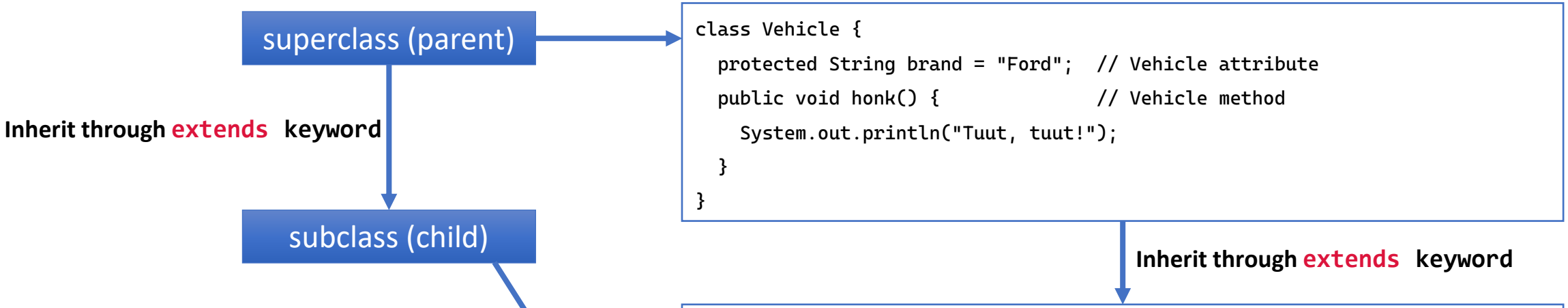


To access Method

Java Inheritance

Java Inheritance

To inherit attributes and methods from one class to another



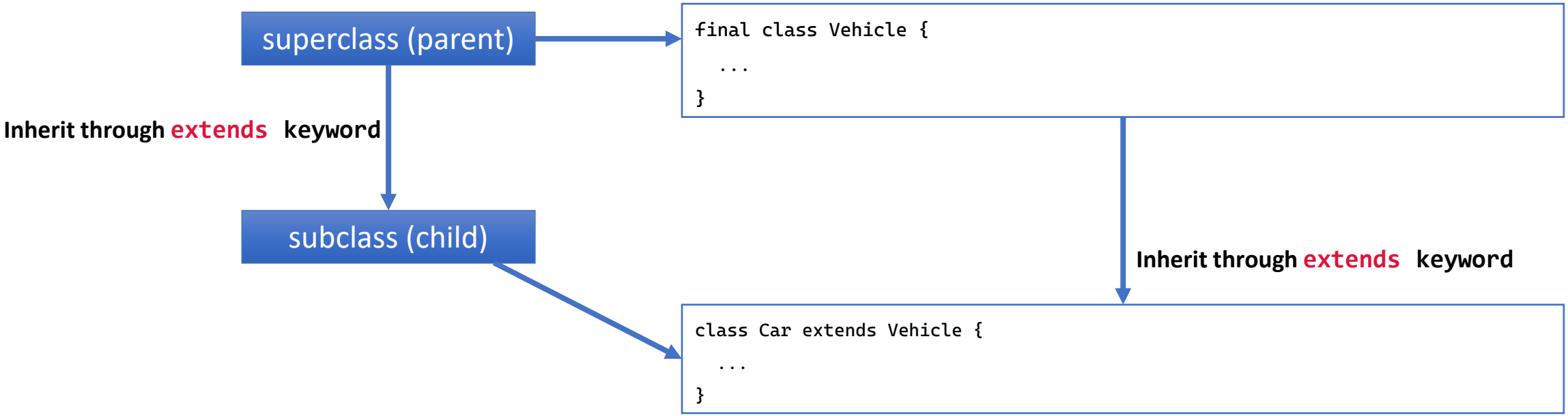

```
class Vehicle {
  protected String brand = "Ford";  // Vehicle attribute
  public void honk() {              // Vehicle method
    System.out.println("Tuut, tuut!");
  }
}
```

Inherit through extends keyword

```
class Car extends Vehicle {
  private String modelName = "Mustang";    // Car attribute
  public static void main(String[] args) {

    // Create a myCar object
    Car myCar = new Car();

    // Call the honk() method (from the Vehicle class) on the myCar object
    myCar.honk();

    // Display the value of the brand attribute (from the Vehicle class) and the
value of the modelName from the Car class
    System.out.println(myCar.brand + " " + myCar.modelName);
  }
}
```

Java Inheritance [final Keyword]

If you don't want other classes to inherit from a class, use the *final* keyword



If you try to access a final class, Java will generate an error

Java Polymorphism

Java Polymorphism

Do you Remember

Inheritance lets us inherit **attributes** and **methods** from another class

Polymorphism uses those **methods** to perform different tasks.

This allows us to perform a single action in different ways.

```
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}
class Pig extends Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
}
class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }
}
class Main {
  public static void main(String[] args) {
    Animal myAnimal = new Animal();  // Create a Animal
object
    Animal myPig = new Pig();  // Create a Pig object
    Animal myDog = new Dog();  // Create a Dog object
    myAnimal.animalSound();
    myPig.animalSound();
    myDog.animalSound();
  }
}
```

Java Inner Class

Java Inner Classes

A class within a class (**Nested classes**)

Purpose of nested classes is to group classes that belong together, which makes your code more readable and maintainable.

```
class OuterClass {
  int x = 10;

  class InnerClass {
    int y = 5;
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);
  }
}

// Outputs 15 (5 + 10)
```

Inner Class

Outer Class

Way to access through object

Do you Remember How to Declare Object ?

```
Class_Name Object_Name = new Class_Name();
Person     myObj       = new Person();
```

```
OuterClass.InnerClass myInner = myOuter.new InnerClass();
```

Outer Class Name

Inner Class Name

Inner Class Object

Outer Class Object

Inner Class Name

Java Inner Classes (Private Inner Class)

Unlike a "regular" class, an **inner class** can be **private** or **protected**.

When use **Private ?**

If you don't want **outside objects** to access the inner class, declare the class as **private**

```
class OuterClass {
  int x = 10;

  private class InnerClass {
    int y = 5;
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);
  }
}
```

If you try to access a private inner class from an outside class, an error occurs

Output:

```
Main.java:13: error: OuterClass.InnerClass has private access
in OuterClass
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
                                            ^
```

Java Inner Classes (Static Inner Class)

An **Static Inner Class** can access without creating an object of the outer class

Note:

just like **static** attributes and methods, a static inner class does not have access to members of the outer class.

Creating InnerClass Object without creating OuterClass Object

Normally we need to create OuterClass **object** first to create InnerClass Object. But Here InnerClass declare here as a **Static** . That's we don't need to create OuterClass object to create InnerClass Object

```
class OuterClass {
  int x = 10;

  static class InnerClass {
    int y = 5;
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass.InnerClass myInner = new OuterClass.InnerClass();
    System.out.println(myInner.y);
  }
}
```

Here we don't create any OuterClass Object To access InnerClass

Output:

5

Java Inner Classes (Access Outer Class From Inner Class)

One advantage of inner classes,

can access **attributes** and **methods** of the outer class

Here InnerClass access the OuterClass **attributes** directly without creating OuterClasss object.

```
class OuterClass {
  int x = 10; // OuterClass attributes

  class InnerClass {
    public int myInnerMethod() {
      return x;  //Access OuterClasss attributes without
                 // creating object
    }
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.myInnerMethod());
  }
}
```

Output:

```
10
```

Java Abstract Class and Method

Java Abstraction

What is Abstraction ?

Data abstraction is the process of **hiding** certain details and **showing** only essential information to the user

abstract keyword : non-access modifier, used for classes and methods

Achieved through

- abstract classes
- interfaces

Abstract class

Is a **restricted class** that cannot be used to create objects

(to access it, it must be inherited from another class)

Abstract method

can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

```
abstract class Animal {

  public abstract void animalSound();

  public void sleep() {

    System.out.println("Zzz");

  }

}
```

it is not possible to create an object of the Animal class

```
Animal myObj = new Animal(); // will generate an error
```

So What is the solution to access the **abstract** Class

Java Abstraction(Via abstraction class and method)

How to access the **abstract** Class

To access the abstract class, it must be inherited from another class.

Inheritance Technique is use

```
// Abstract class
abstract class Animal {
  // Abstract method (does not have a body)
  public abstract void animalSound();
  // Regular method
  public void sleep() {
    System.out.println("Zzz");
  }
}
```

Abstract class

Extending the Animal Class

(inherit from Animal)

```
// Subclass (inherit from Animal)
class Pig extends Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
}
class Main {
  public static void main(String[] args) {
    Pig myPig = new Pig(); // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Subclass (inherit from Animal)

Java Abstraction(Via abstraction class and method)

How to access the **abstract** Class

To access the abstract class, it must be inherited from another class.

Inheritance Technique is use

```
// Abstract class
abstract class Animal {
  // Abstract method (does not have a body)
  public abstract void animalSound();
  // Regular method
  public void sleep() {
    System.out.println("Zzz");
  }
}
```

Abstract class

Extending the Animal Class

(inherit from Animal)

```
// Subclass (inherit from Animal)
class Pig extends Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
}
class Main {
  public static void main(String[] args) {
    Pig myPig = new Pig(); // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Subclass (inherit from Animal)

Why And When To Use Abstract Classes and Methods

To achieve security - hide certain details and only show the important details of an object.

Java Interface

Java Abstraction(Via Interfaces)

Another way to achieve abstraction in Java, is with interfaces

What is Interfaces ? → interface is a completely "abstract class" that is used to group related **methods** with empty bodies

To access the interface methods, the interface must be "implemented" (kinda like inherited) by another class with the implements keyword (instead of extends).

The body of the interface method is provided by the "**implement**" class

```
// interface
interface Animal {
 // interface method (does not have a body)
public void animalSound();
  public void run(); // interface method (does not have a body)
}
```

```
// Pig "implements" the Animal interface
class Pig implements Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
  public void sleep() {
    // The body of sleep() is provided here
    System.out.println("Zzz");
  }
}
class Main {
  public static void main(String[] args) {
    Pig myPig = new Pig();  // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Method Body Descrive here

Java Abstraction(Important notes on Interfaces)

- Like **abstract classes**, interfaces **cannot** be used to create objects.
- Interface methods do not have a body - the body is provided by the "implement" class
- Interface **methods** are by default abstract and public
- Interface **attributes** are by default public, static and final
- An interface cannot contain a constructor (as it cannot be used to create objects)
- On implementation of an interface, you must override all of its methods

```
// interface
interface Animal {
 // interface method (does not have a body)
public void animalSound();
public void run(); // interface method (does not have a body)
}
```

```
// Pig "implements" the Animal interface
class Pig implements Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
  public void sleep() {
    // The body of sleep() is provided here
    System.out.println("Zzz");
  }
}
class Main {
  public static void main(String[] args) {
    Pig myPig = new Pig();  // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Java Abstraction(Multiple Interfaces)

- To implement multiple interfaces, separate them with a comma

Why And When To Use Interfaces?

1. To achieve security - hide certain details and only show the important details of an object (interface).
2. Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can implement multiple interfaces. Note: To implement multiple interfaces, separate them with a comma

```
interface FirstInterface {
  public void myMethod(); // interface method
}
interface SecondInterface {
  public void myOtherMethod(); // interface method
}

class DemoClass implements FirstInterface, SecondInterface {
  public void myMethod() {
    System.out.println("Some text..");
  }
  public void myOtherMethod() {
    System.out.println("Some other text...");
  }
}
```

```
class Main {
  public static void main(String[] args) {
    DemoClass myObj = new DemoClass();
    myObj.myMethod();
    myObj.myOtherMethod();
  }
}
```

This Keyword

this Keyword

Refer Current Class instance Variable

```
class Person {
String name;
int age;
//Constructor
Person ( String name , int age )
{
this.name = name;
this.age = age;
}
}
```

Class level Variable / Instance Variable

Local Variable

Here **this** keyword refer to the Person Class Variable

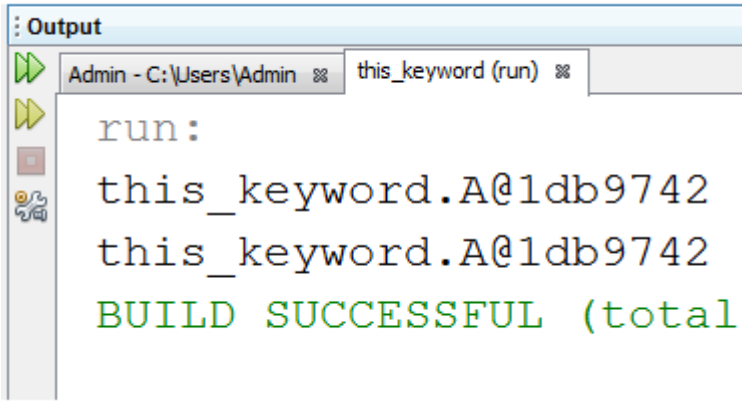
Uses of "**this**" Keyword

- Refer Current Class instance Variable
- It can be used to call current class constructor
- It can be used to call current class method
- It can be passed as an argument in the method (event handling)

this Keyword

1. যেভাবে কোনো অবজেক্ট ক্লাসের সকল উপাদান যেমন ফাংশন এবং কন্সট্রাক্টরকে নির্দেশ করে ঠিক তেমনি "this" কীওয়ার্ড ঐ অবজেক্ট কে নির্দেশ করে।

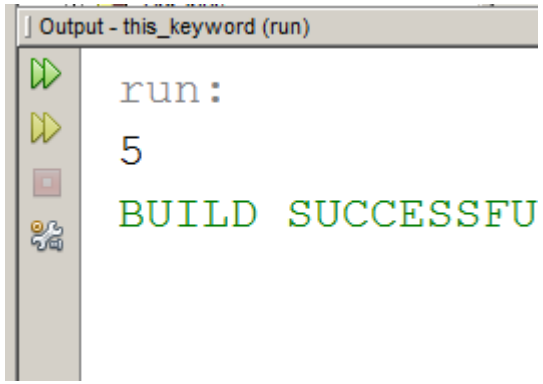
```
public class A {
    void dis()
    {
        System.out.println(this);
    }

}
class B{
     public static void main(String[] args) {
         A res=new A();
         System.out.println(res);
         res.dis();


    }
}
```



এখানে res নামে যে অবজেক্টি তৈরী করা হয়েছে তা থেকে যা আউটপুট আসছে ঠিক তেমনি dis(); ফাংশন থেকে তাই আউটপুট আসছে।

2. ইনস্ট্যান্স এবং লোকাল ভ্যারিয়েবল যদি একই থাকে তবে জাভার কম্পাইলার তা বুজতে পারেনা। তখন "this" কীওয়ার্ড ঐ ভ্যারিয়েবলের আগে বসিয়ে দিলে জাভার কম্পাইলার বুজতে পারে ওটি ইনস্ট্যান্স ভ্যারিয়েবল।

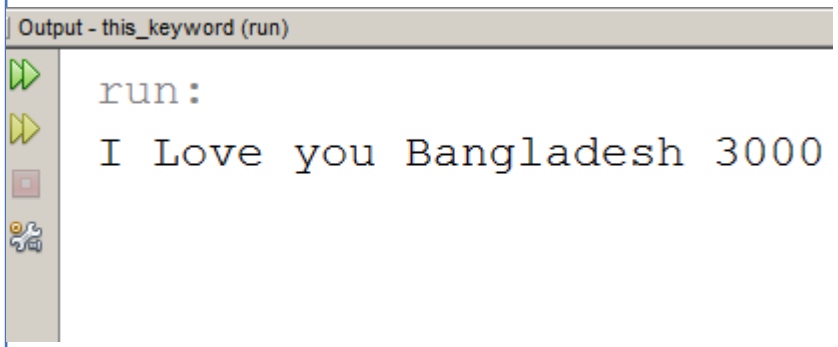
```
public class A {
    int x; // ইনস্ট্যান্স ভ্যারিয়েবল
    A (int x) // int x ->লোকাল ভ্যারিয়েবল এবং এটি একটি ডিফল্ট
কন্সট্রাক্টর
    {
        this.x=x; // this.x -> ইনস্ট্যান্স ভ্যারিয়েবল
    }
    void dis()
    {
        System.out.println(x);
    }
}
class B{
     public static void main(String[] args) {
         A res=new A(5); // অবজেক্ট তৈরি হতেই ডিফল্ট কন্সট্রাক্টরকে
কল করবে।
         res.dis();


    }
}
```



এখানে ইনস্ট্যান্স এবং লোকাল ভ্যারিয়েবল একই তাই ইনস্ট্যান্স ভ্যারিয়েবলকে বুজতে এর আগে ডট "this" বসিয়ে দেওয়া হয়।

3. অবজেক্ট ছাড়া ডিফল্ট কন্সট্রাক্টরকে কল করতে this কীওয়ার্ড ব্যবহার করতে হয়।

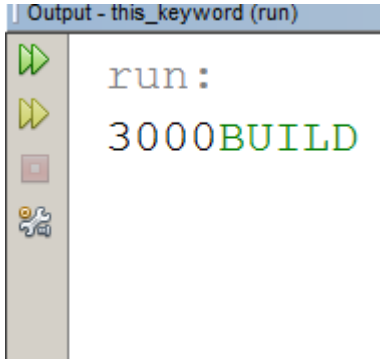
```
public class A {

    A ()
    {
       System.out.print("I Love you Bangladesh");
    }
    A (int x)
    {
        this();
       System.out.print(" "+x+"   ");
    }


}
class B{
     public static void main(String[] args) {
         A res=new A(3000);

    }
}
```



এখানে প্যারামিটার কন্সট্রাক্টরের ভিতরে this(); ফাংশন ব্যবহার করা হয়েছে। প্রথমে মেইন ফাংশন থেকে যখন একটি অবজেক্ট তৈরী করা হয় প্যারামিটার টাইপ তখন তা ক্লাসের সাথে সাথে ক্লাস A এর প্যারামিটার টাইপ কন্সট্রাক্টরকে কল করবে। তখন this(); ফাংশনটি দেখার সাথে সাথে উপরের কন্সট্রাক্টরকে কল করবে।

4. অবজেক্ট ছাড়া প্যারামিটার কন্সট্রাক্টরকে কল করতে this কীওয়ার্ড ব্যবহার করা যায়।

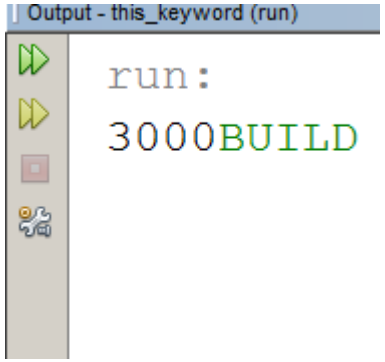
```
public class A {

    A ()
    {
      this(3000);
    }
    A (int x)
    {
       System.out.print(x);
    }



}
class B{
     public static void main(String[] args) {
         A res=new A();

    }
}
```



এখানে যখন অবজেক্ট তৈরি হয়েছে তখন সেটি কন্সট্রাক্টরকে কল করে সেখানে this(3000) প্যারামিটার কন্সট্রাক্টরকে কল করে।

4. অবজেক্ট ছাড়া প্যারামিটার কন্সট্রাক্টরকে কল করতে this কীওয়ার্ড ব্যবহার করা যায়।

```
public class A {

    A ()
    {
      this(3000);
    }
    A (int x)
    {
       System.out.print(x);
    }



}
class B{
     public static void main(String[] args) {
         A res=new A();

    }
}
```



এখানে যখন অবজেক্ট তৈরি হয়েছে তখন সেটি কন্সট্রাক্টরকে কল করে সেখানে this(3000) প্যারামিটার কন্সট্রাক্টরকে কল করে।

Super Keyword

Super Keyword

4.

```
class Animal { // Superclass (parent)
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}

class Dog extends Animal { // Subclass (child)
  public void animalSound() {
    super.animalSound(); // Call the superclass method
    System.out.println("The dog says: bow wow");
  }
}

public class Main {
  public static void main(String args[]) {
    Animal myDog = new Dog(); // Create a Dog object
    myDog.animalSound(); // Call the method on the Dog object
  }
}
```

/

Output:

```
The animal makes a sound
The dog says: bow wow
```

Anonymous Object

Anonymous Object in Java – Part1

Example Code

```
import java.io.*;

class Person {
            String name;
            int age;

            Person(String name, int age)
            {
                        this.name = name;
                        this.age = age;
            }

            void display()
            {
            System.out.println("Name: " + name

               + ", Age: " + age);
            }
}

public class Main {
            public static void main(String[] args)
            {
            // Create an anonymous object
            // and call its display method
            new Person("John", 30).display();
            }
}
```

object **without a name.**

Output:

```
Name: John, Age: 30
```

In Java, an anonymous object is an object that is created **without giving it a name**.

Anonymous objects are often used to create objects on the fly and pass them as arguments to methods. Here is an example of how to create and use an anonymous object in Java.

Anonymous Object in Java – Part2

Example Code

```
import java.io.*;

class OuterClass {
            class InnerClass {
            void display()
            {
            System.out.println("Inside InnerClass");
            }
}

public static void main(String[] args)
            {
// Create an anonymous object of the InnerClass and
// call its display method
new OuterClass().new InnerClass().display();

}
}
```

object **without a name.**

Inside the main method, we create an anonymous object of the InnerClass using the new keyword and the OuterClass.new InnerClass() syntax. This creates a new instance of the InnerClass and assigns it to an anonymous object. We then call the display method on the anonymous object using the dot notation (e.g. obj.display()). This calls the display method on the anonymous object and prints the message “Inside InnerClass” to the console.

Output:

```
Name: John, Age: 30
```

Anonymous Class

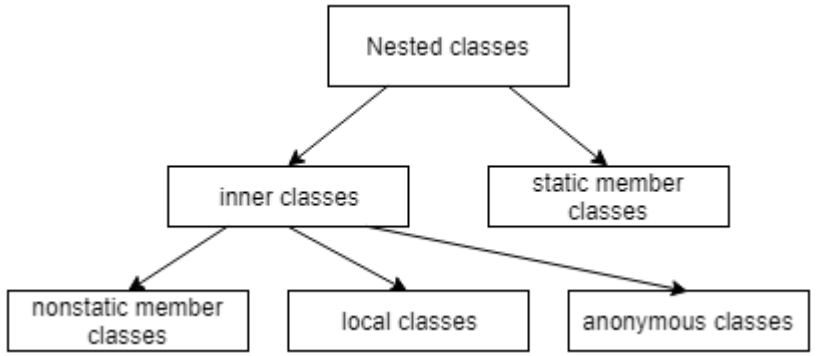
Anonymous Class in Java – Part1

Example Code

```
class Polygon {
    public void display() {
        System.out.println("Inside the Polygon class");
    }
}

class AnonymousDemo {
    public void createClass() {

        // creation of anonymous class extending class Polygon
        Polygon p1 = new Polygon() {
            public void display() {
                System.out.println("Inside an anonymous class.");
            }
        };
        p1.display();
    }
}

class Main {
    public static void main(String[] args) {
        AnonymousDemo an = new AnonymousDemo();
        an.createClass();
    }
}
```

A nested class that doesn't have any name is known as an anonymous class.
An anonymous class must be defined inside another class. Hence, it is also known as an anonymous inner class. Its syntax is:

```
class outerClass {

    // defining anonymous class
    object1 = new Type(parameterList) {
        // body of the anonymous class
    };
}
```



Since they have no name, we can't use them in order to create instances of anonymous classes

Output:

```
Name: John, Age: 30
```

Anonymous Class in Java – Part1

Example Code

```
public class Test {
    public static void main(String[] args) {
     A obj = new A(){
        public void print1(){

            System.out.println("Overrided through annonymous class");
            display_mthd();
        }
        public void display_mthd(){
            System.out.println("dd");
        }
      };
obj.print1();
obj.print2();
obj.print3();
//obj. display_mthd(); //This will create an error
A obj2 = new A();
obj2.print1();
}
}
class A {

    public void print1() {

        System.out.println("Mehtod1");
    }
    public void print2() {

        System.out.println("Mehtod2");
    }

    public void print3() {

        System.out.println("Mehtod3");
    }
}
```

Create the Object where there is no class name. This is called anonymous class. Class is created but there is no class name and also extend the Class A .This anonymous class Override the print1() method.

Can not call display_mthd() method through the anonymous object. Can use only inside the classs.Here obj is the anonymous class object.

Output:

Anonymous Class in Case of interface

```java
package anonymousClass;

public class Test {
    public static void main(String[] args) {

        B obj = new B();
        obj.print1();

    }
}

interface A {
    void print1();
    void print2();
}

class B implements A {
    public void print1() {
        System.out.println("Dipta");
    }
    public void print2() {
        System.out.println("AUST");
    }
}
```

Creating Annonymous Class.পুরো এতটুকুর পরিবর্তে এই এতটুকু দিয়ে কাজ করা যাবে।

```java
package anonymousClass;

public class Test {
    public static void main(String[] args) {

        A obj = new A(){

        };

    }
}

interface A {
    void print1();
    void print2();
}

class B implements A {
    public void print1() {
        System.out.println("Dipta");
    }
    public void print2() {
        System.out.println("AUST");
    }
}
```

Test3.java > ...

```java
class Test3 {
    Run | Debug
    public static void main(String[] args) {

        A obj = new A(){
            public void print1(){
                System.out.println(x:"A");
            }
        };
        obj.print1();
    }
}

interface A {

    void print1();
    //void print2();

}
```

এখানে interface কে implements করা হয়েছে , anonymous class এর সাহায্যে। কতগুলো লাইন কমে গেল এতে।

Anonymous Class in Case of abstract Class

Anonymous Class in Case of abstract Class