**‘This’ Keyword in Java**

**Definition:**

The this keyword in Java refers to the current instance of a class. It is used to differentiate between class fields (instance variables) and parameters that have the same name. It can also be used to call the constructor of the class, refer to instance methods, and access the current object.

**Uses of this Keyword:**

1. **Accessing instance variables**: It is used when the parameter name is the same as the instance variable name.
2. **Calling instance methods**: It can be used to call other methods in the current class.
3. **Calling the constructor**: It can be used to call another constructor within the same class.
4. **Referring to the current object**: It can be used to refer to the current object.

**Example:**

class Student {

// Instance variables

String name;

int age;

// Constructor with parameters

public Student(String name, int age) {

// Using 'this' to differentiate between instance variables and parameters

this.name = name;

this.age = age;

}

// Method to display details of the student

public void display() {

// Using 'this' to refer to the current instance of the object

System.out.println("Name: " + this.name);

System.out.println("Age: " + this.age);

}

public static void main(String[] args) {

// Creating an object of Student class

Student student1 = new Student("John", 20);

// Calling the display method

student1.display();

}

}

### The static Keyword in Java

#### Definition:

The static keyword in Java is used to indicate that a particular member (variable, method, or block) belongs to the class itself rather than to instances of the class. It can be applied to variables, methods, blocks, and inner classes.

#### Uses of static Keyword:

1. **Static Variables**: A variable declared as static is shared among all instances of the class. It is created only once and is accessible by all objects of that class.
2. **Static Methods**: A method declared as static can be called without creating an object of the class. Static methods can only access other static members (variables or methods) of the class.
3. **Static Block**: A static block is used for static initializations of a class. It is executed once when the class is first loaded into memory.
4. **Static Classes (Inner Classes)**: A static inner class does not have a reference to the instance of the outer class and can be instantiated without an instance of the outer class.

### Example:

### class Counter {

// Static variable

static int count = 0;

// Static method

public static void increment() {

count++;

}

// Instance method

public void displayCount() {

System.out.println("Count: " + count);

}

public static void main(String[] args) {

// Calling static method without creating an instance of the class

Counter.increment();

Counter.increment();

// Creating an instance to access the instance method

Counter counterObj = new Counter();

counterObj.displayCount(); // Output: Count: 2

}

}

### Explanation:

1. **Static Variable (count)**: The variable count is static, so it is shared across all instances of the Counter class. Every time the increment() method is called, the value of count increases.
2. **Static Method (increment())**: The increment() method is static and can be called without creating an instance of the Counter class.
3. **Instance Method (displayCount())**: The displayCount() method is an instance method and needs an object to be called. It displays the value of the static count variable.

### Key Points:

* Static members are shared among all objects of the class.
* Static methods can access only static variables and other static methods.
* Static members are initialized when the class is loaded for the first time.

### The super Keyword in Java

#### Definition:

The super keyword in Java is used to refer to the immediate parent class (superclass) of the current object. It is used to access members (fields, methods, constructors) of the parent class.

#### Uses of super Keyword:

1. **Accessing Parent Class Constructor**: The super() is used to call the constructor of the parent class from a subclass constructor.
2. **Accessing Parent Class Methods**: The super keyword can be used to invoke a method in the parent class, especially when the subclass has overridden the method.
3. **Accessing Parent Class Fields**: It can be used to access the instance variables of the parent class, particularly when the subclass has fields with the same name.

### Example:

class Animal {

// Constructor of Animal class

public Animal() {

System.out.println("Animal class constructor");

}

// Method in the Animal class

public void speak() {

System.out.println("Animal speaks");

}

}

class Dog extends Animal {

// Constructor of Dog class

public Dog() {

// Calling parent class constructor

super();

System.out.println("Dog class constructor");

}

// Overriding speak method

@Override

public void speak() {

// Calling parent class method using super

super.speak();

System.out.println("Dog barks");

}

public static void main(String[] args) {

Dog dog = new Dog();

dog.speak(); // Output: Animal speaks \n Dog barks

}

}

### Key Points:

* super can be used to call the constructor of the parent class.
* It is used to invoke methods of the parent class, even if they are overridden by the subclass.
* It can be used to refer to fields in the parent class that may be shadowed by the subclass.

### ****Inner Class in Java****

#### ****Definition:****

An **inner class** is a class that is defined within another class. It can access the members (variables, methods) of the outer class, including private ones. Inner classes are useful when you need to logically group classes together or if the inner class is only useful in the context of the outer class.

#### ****Types of Inner Classes:****

1. **Member Inner Class**: Defined inside the outer class but outside any methods.
2. **Local Inner Class**: Defined inside a method or a constructor of the outer class.
3. **Anonymous Inner Class**: A nameless class that is defined and instantiated at the same time.
4. **Static Nested Class**: Defined inside the outer class with the static modifier (this is a type of nested class, not a standard inner class).

#### ****Example Program: Member Inner Class****

class Outer {

private String outerMessage = "Hello from Outer class";

// Member Inner Class

class Inner {

public void display() {

System.out.println(outerMessage); // Accessing outer class member

}

}

public void createInner() {

Inner inner = new Inner();

inner.display();

}

public static void main(String[] args) {

Outer outer = new Outer();

outer.createInner(); // Creates an instance of Inner class and calls display

}

}

### ****Working Procedure:****

* The **Inner** class is defined within the **Outer** class.
* To access the inner class, you need an instance of the outer class first.
* The inner class can directly access the members of the outer class, including private members.

**Output:**

csharp

Copy code

Hello from Outer class

### ****Nested Class in Java****

#### ****Definition:****

A **nested class** is a class defined inside another class. Unlike inner classes, nested classes can be either static or non-static. A **static nested class** does not require an instance of the outer class, while a **non-static nested class** (inner class) requires an instance of the outer class.

#### ****Types of Nested Classes:****

1. **Static Nested Class**: A static class defined within another class, which does not have access to the outer class’s instance variables or methods unless they are also static.
2. **Non-static Nested Class (Inner Class)**: An instance class defined within another class, which has access to both static and non-static members of the outer class.

#### ****Example Program: Static Nested Class****

class Outer {

private static String outerMessage = "Hello from Static Nested Class";

// Static Nested Class

static class Nested {

public void display() {

System.out.println(outerMessage); // Accessing static member of outer class

}

}

public static void main(String[] args) {

Outer.Nested nested = new Outer.Nested(); // No need for an instance of Outer class

nested.display(); // Calls the display method of the static nested class

}

}

### ****Working Procedure:****

* The **Nested** class is a static class, so it can be instantiated without an instance of the outer class.
* The static nested class can only access static members of the outer class.
* To access the static nested class, you refer to it using the outer class name (e.g., Outer.Nested).

**Output:**

Hello from Static Nested Class

### ****Key Differences Between Inner Class and Nested Class:****

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Inner Class** | **Nested Class** |
| **Definition** | A non-static class inside another class | Any class inside another class |
| **Access to Outer Class Members** | Can access both static and non-static members | Static nested class can only access static members |
| **Instance Requirement** | Requires an instance of the outer class | Static nested class doesn’t need an instance of the outer class |
| **Instantiation** | Requires outer class instance to instantiate | Static nested class can be instantiated without outer class instance |

### ****Java Command Line Arguments:****

#### ****Definition:****

#### Java command-line argument is an argument i.e. passed at the time of running the Java program. In Java, the command line arguments passed from the console can be received in the Java program and they can be used as input. The users can pass the arguments during the execution bypassing the command-line arguments inside the main() method.

#### Working command-line arguments

#### We need to pass the arguments as space-separated values. We can pass both strings and primitive data types(int, double, float, char, etc) as command-line arguments. These arguments convert into a string array and are provided to the main() function as a string array argument. When command-line arguments are supplied to JVM, JVM wraps these and supplies them to args[]. It can be confirmed that they are wrapped up in an args array by checking the length of args using args.length. Internally, JVM wraps up these command-line arguments into the args[ ] array that we pass into the main() function. We can check these arguments using args.length method. JVM stores the first command-line argument at args[0], the second at args[1], the third at args[2], and so on.****Syntax:****

public static void main(String[] args)

* The args parameter is an array of String that holds the command-line arguments.
* Each argument is a space-separated string, and you can access them by index (args[0], args[1], etc.).

#### ****Example Program with 6 Command Line Arguments:****

public class CommandLineExample {

public static void main(String[] args) {

// Check if exactly 6 arguments are passed

if (args.length == 6) {

System.out.println("The command line arguments are:");

for (int i = 0; i < args.length; i++) {

System.out.println("Argument " + (i + 1) + ": " + args[i]);

}

} else {

System.out.println("Please pass exactly 6 arguments.");

}

}

}

#### ****How to Run the Program:****

1. Compile the Java program:

javac CommandLineExample.java

1. Run the program with 6 arguments:

java CommandLineExample 10 20 30 40 50 60

#### ****Output:****

The command line arguments are:

Argument 1: 10

Argument 2: 20

Argument 3: 30

Argument 4: 40

Argument 5: 50

Argument 6: 60

#### ****Explanation:****

* The program checks if exactly 6 arguments are passed. If so, it prints each argument one by one.
* The args array stores each argument as a String.
* You can manipulate or convert these arguments into other data types (e.g., integers) if needed using Integer.parseInt() or other conversion methods.

### ****Key Points:****

* Command-line arguments are always passed as strings.
* You can pass multiple arguments when running the program, and they are indexed starting from args[0].
* You can check the length of the args array to ensure that the correct number of arguments is provided.

### ****Java Wrapper Class****

#### ****Definition:****

#### A Wrapper class in Java is a class whose object wraps or contains primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types. In other words, we can wrap a primitive value into a wrapper class object. Let’s check on the wrapper classes in Java with examples:

#### Need of Wrapper Classes

#### There are certain needs for using the Wrapper class in Java as mentioned below:

#### They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).

#### The classes in java.util package handles only objects and hence wrapper classes help in this case also.

#### Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.

#### An object is needed to support synchronization in multithreading

#### ****Wrapper Classes for Primitive Data Types:****

|  |  |
| --- | --- |
| **Primitive Type** | **Wrapper Class** |
| int | Integer |
| char | Character |
| boolean | Boolean |
| byte | Byte |
| short | Short |
| long | Long |
| float | Float |
| double | Double |

#### ****Common Uses of Wrapper Classes:****

1. **Conversion between primitive types and objects**: Wrapper classes allow you to convert primitive types to objects and vice versa. This is useful when interacting with classes and collections that only accept objects.
2. **Utility methods**: Wrapper classes provide utility methods like parseInt(), valueOf(), toString(), etc., for converting between different types.

#### ****Example Program: Using Wrapper Class****

public class WrapperExample {

public static void main(String[] args) {

// Convert primitive to wrapper class object (Autoboxing)

int primitiveInt = 10;

Integer wrapperInt = primitiveInt; // Autoboxing

// Convert wrapper class object to primitive (Unboxing)

int unboxedInt = wrapperInt; // Unboxing

System.out.println("Primitive int: " + primitiveInt);

System.out.println("Wrapper Integer: " + wrapperInt);

System.out.println("Unboxed int: " + unboxedInt);

// Using a wrapper class utility method (parseInt)

String numStr = "123";

int parsedInt = Integer.parseInt(numStr);

System.out.println("Parsed integer: " + parsedInt);

}

}

#### ****Output:****

Primitive int: 10

Wrapper Integer: 10

Unboxed int: 10

Parsed integer: 123

#### ****Explanation:****

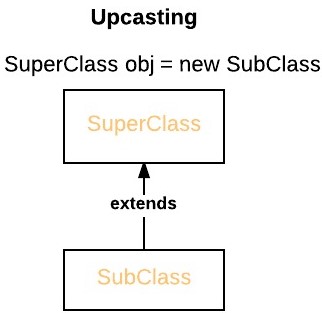
* **Autoboxing**: The primitive value 10 is automatically converted to the Integer object when assigned to wrapperInt.
* **Unboxing**: The Integer object is automatically converted back to the primitive int when assigned to unboxedInt.
* **Utility Methods**: The Integer.parseInt() method is used to convert a String representation of a number into an int.

### ****Key Points:****

1. **Autoboxing** and **unboxing**: Java automatically converts between primitives and their corresponding wrapper objects.
2. **Wrapper Classes** provide methods to manipulate primitive values as objects.
3. They are essential when working with data structures like ArrayList, as they require objects instead of primitive types.

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

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

When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.

At run-time, it depends on the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed

A superclass reference variable can refer to a subclass object. This is also known as upcasting. Java uses this fact to resolve calls to overridden methods at run time. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed. Here is an example that illustrates dynamic method dispatch:

class Class\_A {

void demo() {

Class\_A a = new Class\_A();

a.demo();

// Upcasting

Class\_A b = new Class\_B();

b.demo();

a = b; // Reference reassignment

a.demo();

}

}

System.out.println("The Class\_A method called");

}

}

class Class\_B extends Class\_A {

@Override

void demo() {

System.out.println("The Class\_B method called");

}

}

public class Main {

public static void main(String[] args) {

**User-Defined Packages in Java**

Packages in Java are a mechanism to encapsulate a group of classes, interfaces, and sub-packages. In Java, it is used for making search/locating and usage of classes, interfaces, enumerations, and annotations easier. It can be considered data encapsulation also. In other words, we can say a package is a container of a group of related classes where some of the classes are accessible are exposed, and others are kept for internal purposes.

Types of Packages

Packages are categorized into two parts. These are:

* Built-in packages: The already defined package like java.io.\*, java. lang.\* etc., are known as built-in packages.
* User-defined packages: As the name propose, user-defined packages in Java are essentially packages that are defined by the programmer. Whenever we want to add a class to the package, we have to mention the package name and the “package” keyword at the top of the program.
* User-defined Packages are those packages that are designed or created by the developer to categorize classes and packages. They are much similar to the built-in that java offers. It can be imported into other classes and used the same as we use built-in packages. But If we omit the package statement, the class names are put into the default package, which has no name.
* Benefits of User-Defined Packages

**Code Organization:** User-defined packages allow developers to group related classes and resources together, making it easier to navigate and understand the codebase. This organization promotes code modularity and improves overall maintainability.

**Encapsulation:** Packages provide a level of encapsulation by allowing you to specify the access level of classes and resources within the package. This helps in controlling the visibility and accessibility of code elements, making it easier to define the boundaries of your code.

**Code Reusability:** By creating user-defined packages, you can encapsulate commonly used classes, utilities, or modules, making them easily reusable across different projects. This reusability saves development time and effort, as well as promotes consistency across applications. To create a package, we’re supposed to use the package keyword

package example;

public class gfg {

    public void show()

    {

        System.out.println("Hello geeks!! How are you?");

    }

    public static void main(String args[])

    {

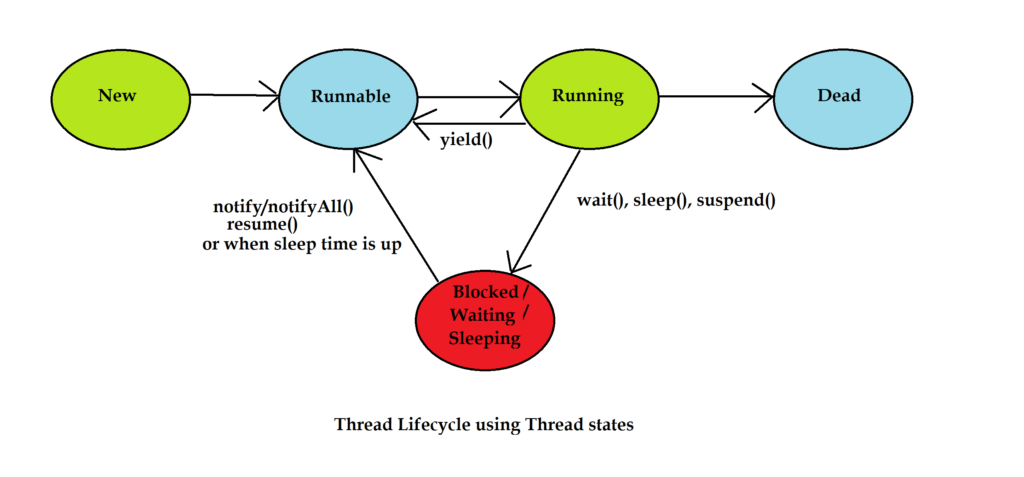
        gfg obj = new gfg();

        obj.show();

    }

}

**Lifecycle and States of a Thread in Java**

A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

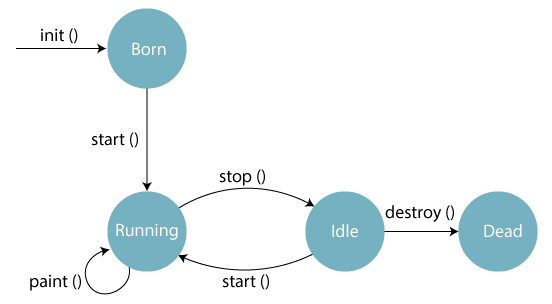
1. New State
2. Runnable State
3. Blocked State
4. Waiting State
5. Timed Waiting State
6. Terminated State

**Life Cycle of a Thread**

There are multiple states of the thread in a lifecycle as mentioned below:

1. **New Thread:** When a new thread is created, it is in the new state. The thread has not yet started to run when the thread is in this state. When a thread lies in the new state, its code is yet to be run and hasn’t started to execute.
2. **Runnable State:** A thread that is ready to run is moved to a runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run. A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lie in a runnable state.
3. **Blocked:** The thread will be in blocked state when it is trying to acquire a lock but currently the lock is acquired by the other thread. The thread will move from the blocked state to runnable state when it acquires the lock.
4. **Waiting state**: The thread will be in waiting state when it calls wait() method or join() method. It will move to the runnable state when other thread will notify or that thread will be terminated.
5. **Timed Waiting:** A thread lies in a timed waiting state when it calls a method with a time-out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.
6. **Terminated State**: A thread terminates because of either of the following reasons:
   1. Because it exits normally. This happens when the code of the thread has been entirely executed by the program.
   2. Because there occurred some unusual erroneous event, like a segmentation fault or an unhandled exception.

# **Applet Life Cycle in Java**

In Java, an applet is a special type of program embedded in the web page to generate dynamic content. Applet is a class in Java.

The applet life cycle can be defined as the process of how the object is created, started, stopped, and destroyed during the entire execution of its application. It basically has five core methods namely init(), start(), stop(), paint() and destroy().These methods are invoked by the browser to execute.

Along with the browser, the applet also works on the client side, thus having less processing time.

## **Methods of Applet Life Cycle**

* **init():** The init() method is the first method to run that initializes the applet. It can be invoked only once at the time of initialization. The web browser creates the initialized objects, i.e., the web browser (after checking the security settings) runs the init() method within the applet.
* **start():** The start() method contains the actual code of the applet and starts the applet. It is invoked immediately after the init() method is invoked. Every time the browser is loaded or refreshed, the start() method is invoked. It is also invoked whenever the applet is maximized, restored, or moving from one tab to another in the browser. It is in an inactive state until the init() method is invoked.
* **stop():** The stop() method stops the execution of the applet. The stop () method is invoked whenever the applet is stopped, minimized, or moving from one tab to another in the browser, the stop() method is invoked. When we go back to that page, the start() method is invoked again.
* **destroy():** The destroy() method destroys the applet after its work is done. It is invoked when the applet window is closed or when the tab containing the webpage is closed. It removes the applet object from memory and is executed only once. We cannot start the applet once it is destroyed.
* **paint():** The paint() method belongs to the Graphics class in Java. It is used to draw shapes like circle, square, trapezium, etc., in the applet. It is executed after the start() method and when the browser or applet windows are resized.

## **CODE :**

**class** TestAppletLifeCycle **extends** Applet {

**public** **void** init() {

// initialized objects

}

**public** **void** start() {

// code to start the applet

}

**public** **void** paint(Graphics graphics) {

// draw the shapes

}

**public** **void** stop() {

// code to stop the applet

}

**public** **void** destroy() {

// code to destroy the applet

}

}

**Sequence of method execution when an applet is executed:**

1. init()
2. start()
3. paint()

**Sequence of method execution when an applet is executed:**

1. stop()
2. destroy()

# **Priority of a Thread (Thread Priority)**

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. Note that not only JVM a Java programmer can also assign the priorities of a thread explicitly in a Java program.

## **Setter & Getter Method of Thread Priority**

Let's discuss the setter and getter method of the thread priority.

**public final int getPriority():** The java.lang.Thread.getPriority() method returns the priority of the given thread.

**public final void setPriority(int newPriority):** The java.lang.Thread.setPriority() method updates or assign the priority of the thread to newPriority. The method throws IllegalArgumentException if the value newPriority goes out of the range, which is 1 (minimum) to 10 (maximum).

## **3 constants defined in Thread class:**

1. public static int MIN\_PRIORITY
2. public static int NORM\_PRIORITY
3. public static int MAX\_PRIORITY

Default priority of a thread is 5 (NORM\_PRIORITY). The value of MIN\_PRIORITY is 1 and the value of MAX\_PRIORITY is 10.

### Example of priority of a Thread:

import java.lang.\*;

public class ThreadPriorityExample extends Thread {

public void run() {

System.out.println("Inside the run() method");

}

public static void main(String argvs[]) {

ThreadPriorityExample th1 = new ThreadPriorityExample();

ThreadPriorityExample th2 = new ThreadPriorityExample();

ThreadPriorityExample th3 = new ThreadPriorityExample();

System.out.println("Priority of the thread th1 is : " + th1.getPriority());

System.out.println("Priority of the thread th2 is : " + th2.getPriority());

System.out.println("Priority of the thread th2 is : " + th2.getPriority());

th1.setPriority(6);

th2.setPriority(3);

th3.setPriority(9);

System.out.println("Priority of the thread th1 is : " + th1.getPriority());

System.out.println("Priority of the thread th2 is : " + th2.getPriority());

System.out.println("Priority of the thread th3 is : " + th3.getPriority());

System.out.println("Currently Executing The Thread : " + Thread.currentThread().getName());

System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());

Thread.currentThread().setPriority(10);

System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());

}

}