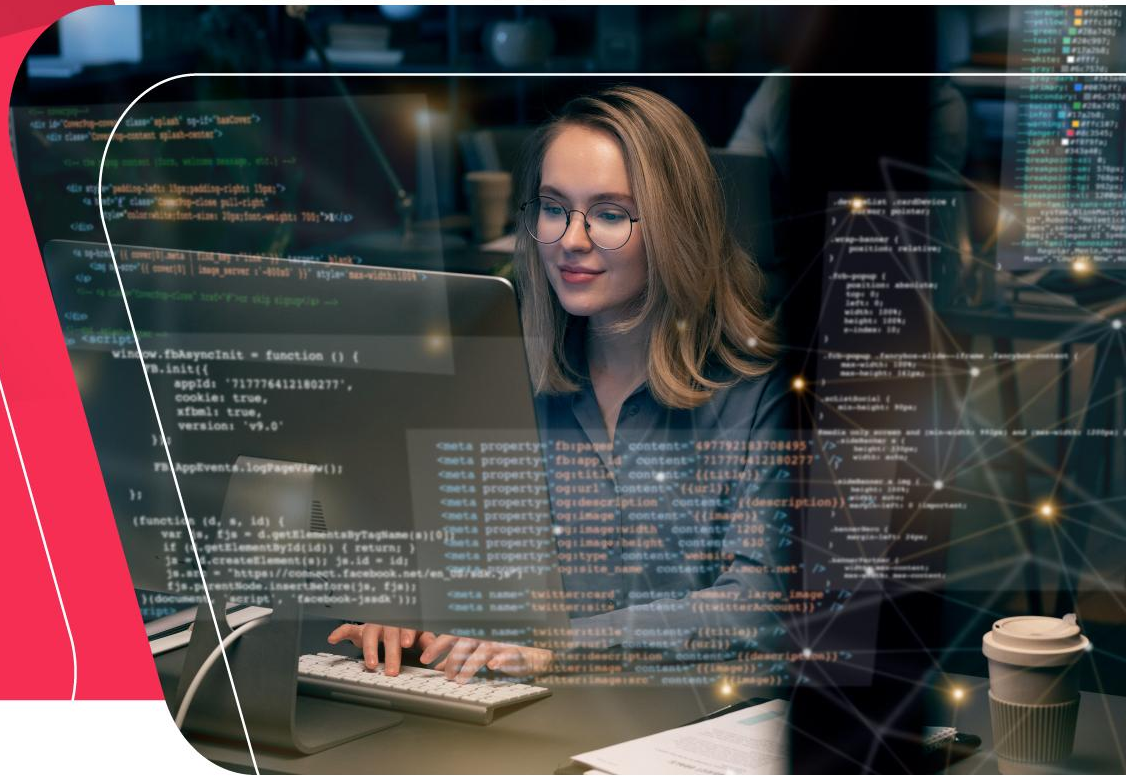
TIA-Academy

Java Bootcamp

Day 13



Technologies will be Use

- JDK 8/**11**/15
- JRE 8/**11**/15
- Writing Code using Notepad++ (For Windows) or Vim (For Linux and Mac)
- Compiling with Command Prompt (For Windows) or Terminal (For Linux and Mac)
- ***Forbidden of using IDE Based Development Tools as of Intellij IDEA or Eclipse***

TIA-Academy

Java Constructors

What is a Constructor?

- A constructor in Java is **similar to a method** that is invoked when an object of the class is created.
- Unlike Java methods, a **constructor has the same name as that of the class** and **does not have any return type**. For example,

```
class Test {
  Test() {
    // constructor body
  }
}
```

- Here, **Test() is a constructor**. It has the same name as that of the class and doesn't have a return type.

What is a Constructor?

- Example 1: Java Constructor
- Output :

```
Constructor Called:
The name is Programiz
```

```
class Main {
  private String name;

  // constructor
  Main() {
    System.out.println("Constructor Called:");
    name = "Programiz";
  }

  public static void main(String[] args) {

    // constructor is invoked while
    // creating an object of the Main class
    Main obj = new Main();
    System.out.println("The name is " + obj.name);
  }
}
```

What is a Constructor?

- In the above example, we have created a **constructor named Main()**. Inside the constructor, we are **initializing the value of the name variable**.
- Notice the statement of creating an object of the Main class.

```
Main obj = new Main();
```

- Here, when the object is created, the **Main() constructor is called**. And, the value of the **name** variable is **initialized**.
- Hence, the program prints the value of the **name variables as Programiz**.

Types of Constructor

In Java, constructors can be divided into 3 types:

1. No-Arg Constructor
2. Parameterized Constructor
3. Default Constructor

Java No-Arg Constructors

- Similar to methods, a **Java constructor may or may not have any parameters** (arguments).
- If a constructor does not accept any parameters, it is known as a **no-argument constructor**. For example,

```
private Constructor() {
    // body of the constructor
}
```

Java No-Arg Constructors

- Example 2: Java private no-arg constructor
- Output :

```
Constructor is called
Value of i: 5
```

```
class Main {
  private String name;

  // constructor
  Main() {
    System.out.println("Constructor Called:");
    name = "Programiz";
  }

  public static void main(String[] args) {

    // constructor is invoked while
    // creating an object of the Main class
    Main obj = new Main();
    System.out.println("The name is " + obj.name);
  }
}
```

Java No-Arg Constructors

- In the above example, we have created a constructor Main(). Here, the constructor does not accept any parameters. Hence, it is known as a no-arg constructor.
- **Notice that we have declared the constructor as private.**
- Once a constructor is declared private**, it cannot be accessed from outside the class**. So, creating objects from outside the class is prohibited using the private constructor.
- Here, we are **creating the object inside the same class. Hence, the program is able to access the constructor.**
- However, if we want to **create objects outside the class**, then we need to declare the constructor as **public**.

Java No-Arg Constructors

- Example 3: Java public no-arg constructors
- Output :

```
Company name = Programiz
```

```
class Company {
  String name;

  // public constructor
  public Company() {
    name = "Programiz";
  }
}

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

    // object is created in another class
    Company obj = new Company();
    System.out.println("Company name = " + obj.name);
  }
}
```

Java Parameterized Constructor

- A Java constructor can also accept one or more parameters.
- Such constructors are known as parameterized constructors **(constructor with parameters).**
- Example 4: Parameterized constructor
- Output :

```
Java Programming Language
Python Programming Language
C Programming Language
```

```
class Main {
  String languages;

  // constructor accepting single value
  Main(String lang) {
    languages = lang;
    System.out.println(languages + " Programming
Language");
  }

  public static void main(String[] args) {

    // call constructor by passing a single value
    Main obj1 = new Main("Java");
    Main obj2 = new Main("Python");
    Main obj3 = new Main("C");
  }
}
```

Java Parameterized Constructor

- In the above example, we have **created a constructor named Main().** Here, the constructor **takes a single parameter**. Notice the expression,

```
Main obj1 = new Main("Java");
```

- Here, we are **passing the single value to the constructor**. Based on the argument passed, the language variable is initialized inside the constructor.

Java Default Constructor

- If we do not create any constructor, the Java compiler automatically create a no-arg constructor during the execution of the program.
- This constructor is called default constructor.
- Example 5: Default Constructor
- Output :

```
a = 0
b = false
```

Here, we haven't created any constructors. Hence, the Java compiler automatically creates the default constructor.

```
class Main {
  int a;
  boolean b;

  public static void main(String[] args) {

    // A default constructor is called
    Main obj = new Main();

    System.out.println("Default Value:");
    System.out.println("a = " + obj.a);
    System.out.println("b = " + obj.b);
  }
}
```

Java Default Constructor

- The default constructor initializes any uninitialized instance variables with default values.

Type	Default Value
`boolean`	false
`byte`	0
`short`	0
`int`	0
`long`	0L
`char`	\u0000
`float`	0.0f
`double`	0.0d
`object`	Reference null

Java Default Constructor

- In the above program, the variables a and b are initialized with default value 0 and false respectively.
- The above program is equivalent to:
- The output of the program is the same as Example 5.

```
class Main {
  int a;
  boolean b;

   Main() {
    a = 0;
    b = false;
  }

  public static void main(String[] args) {
    // call the constructor
    Main obj = new Main();

    System.out.println("Default Value:");
    System.out.println("a = " + obj.a);
    System.out.println("b = " + obj.b);
  }
}
```

Important Notes on Java Constructors

- Constructors are invoked implicitly when you instantiate objects.
- The two rules for creating a constructor are:
  - The name of the constructor should be the same as the class.
  - A Java constructor must not have a return type.
- **If a class doesn't have a constructor, the Java compiler automatically creates a default constructor during run-time**.
- The **default constructor initializes instance variables with default values. For example, the int variable will be initialized to 0**

Important Notes on Java Constructors

- Constructor types:
  - **No-Arg Constructor** - a constructor that does not accept any arguments
  - **Parameterized constructor** - a constructor that accepts arguments
  - **Default Constructor** - a constructor that is automatically created by the Java compiler if it is not explicitly defined.
- A constructor cannot be abstract or static or final.
- A constructor can be overloaded but can not be overridden.

Constructors Overloading in Java

- Similar to Java method overloading, we can also create two or more constructors with different parameters. This is called constructors overloading.
- Example 6: Java Constructor Overloading
- Output:

```
Programming Language: Java
Programming Language: Python
```

```
class Main {
  String language;

  // constructor with no parameter
  Main() {
    this.language = "Java";
  }

  // constructor with a single parameter
  Main(String language) {
    this.language = language;
  }

  public void getName() {
    System.out.println("Programming Langauage: " + this.language);
  }

  public static void main(String[] args) {

    // call constructor with no parameter
    Main obj1 = new Main();

    // call constructor with a single parameter
    Main obj2 = new Main("Python");

    obj1.getName();
    obj2.getName();
  }
}
```

Constructors Overloading in Java

- In the above example, we have two constructors: **Main() and Main(String language).** Here, both the constructor initialize the value of the variable language with **different values.**
- Based on the parameter passed during object creation, different constructors are called and different values are assigned.
- It is also possible to call one constructor from another constructor.

TIA-Academy

ASSIGNMENT 01

TIA-Academy

Java Strings

Java Strings

- In this tutorial, we will learn about Java strings, how to create them, and various methods of the String class with the help of examples.
- In Java, a string is a sequence of characters. For example, **"hello"** is a string containing a sequence of characters **'h', 'e', 'l', 'l', and 'o'.**
- We **use double quotes** to represent a string in Java. For example,

```
// create a string
String type = "Java programming";
```

- Here, we have created a **string variable named type**. The variable is initialized with the string Java Programming.

Java Strings

- Example: Create a String in Java.

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

    // create strings
    String first = "Java";
    String second = "Python";
    String third = "JavaScript";

    // print strings
    System.out.println(first);   // print Java
    System.out.println(second);  // print Python
    System.out.println(third);   // print JavaScript
  }
}
```

Java Strings

- In the above example, we have created three strings named first, second, and third. Here, we are directly creating strings like primitive types.
- However, there is another way of creating Java strings (using the **new** keyword). We will learn about that later in this tutorial.
- Note: **Strings in Java are not primitive types** (like int, char, etc). Instead, all strings are objects of a predefined class named String.
- And, **all string variables are instances of the String class**.

Java Strings Operations

- Java String provides various methods to perform different operations on strings.
- We will look into some of the commonly used string operations.
  1. Get length of a String
  2. Join Two Java Strings
  3. Compare two Strings

Get length of a String

- To find the length of a string, we use the length() method of the String. For example,
- Output :

```
String: Hello! World
Length: 12
```

- In the above example, **the length() method calculates the total number of characters** in the string and returns it.

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

    // create a string
    String greet = "Hello! World";
    System.out.println("String: " + greet);

    // get the length of greet
    int length = greet.length();
    System.out.println("Length: " + length);
  }
}
```

Join Two Java Strings

- We can join two strings in Java using the concat() method. For example,
- Output :

```
First String: Java
Second String: Programming
Joined String: Java Programming
```

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

    // create first string
    String first = "Java ";
    System.out.println("First String: " + first);

    // create second
    String second = "Programming";
    System.out.println("Second String: " + second);

    // join two strings
    String joinedString = first.concat(second);
    System.out.println("Joined String: " + joinedString);
  }
}
```

Join Two Java Strings

- In the above example, we have created two strings named first and second. Notice the statement,

```
String joinedString = first.concat(second);
```

- Here, the concat() method joins the second string to the first string and assigns it to the joinedString variable.
- We can also join two strings using the + operator in Java. To learn more, visit Java String concat().

Compare two Strings

- In Java, we can make comparisons between two strings using the equals() method. For example,
- Output :

```
Strings first and second are equal: true
Strings first and third are equal: false
```

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

    // create 3 strings
    String first = "java programming";
    String second = "java programming";
    String third = "python programming";

    // compare first and second strings
    boolean result1 = first.equals(second);
    System.out.println("Strings first and second are equal: " + result1);

    // compare first and third strings
    boolean result2 = first.equals(third);
    System.out.println("Strings first and third are equal: " + result2);
  }
}
```

Compare two Strings

- In the above example, we have created 3 strings named first, second, and third. Here, we are using the **equal() method to check if one string is equal to another**.
- The **equals() method checks the content of strings while comparing them.**
- Note: We can also compare two strings using the == operator in Java. However, this approach is different than the equals() method

Escape character in Java Strings

- **The escape character is used to escape some of the characters present inside a string.**
- Suppose we need to include double quotes inside a string.

```
// include double quote
String example = "This is the  "String" class";
```

- Since strings are represented by double quotes, the compiler will treat "This is the " as the string. **Hence, the above code will cause an error.**
- To solve this issue, we use the **escape character \ in Java**. For example,

```
// use the escape character
String example = "This is the  \"String\" class.";
```

- Now **escape characters tell the compiler to escape double quotes** and read the whole text.

Java Strings are Immutable

- In Java, strings are immutable. This means, once we create a string, we cannot change that string.
- To understand it more deeply, consider an example:

```
// create a string
String example = "Hello! ";
```

- Here, we have created a **string variable named example**. The **variable holds the string "Hello! ".**
- Now suppose we want to change the string.

```
// add another string "World"
// to the previous string example
example = example.concat(" World");
```

- Here, we are using **the concat() method to add another string World** to the previous string.
- It looks like we are able to change the value of the previous string. However, this is not true.

Java Strings are Immutable

Let's see what has happened here,

1. JVM takes the first string "Hello! "
2. creates a new string by adding "World" to the first string
3. assign the new string "Hello! World" to the example variable
4. the first string "Hello! " remains unchanged

Creating strings using the new keyword

- So far we have created strings like primitive types in Java.
- Since **strings in Java are objects**, we can create strings using the **new** keyword as well. For example,

```
// create a string using the new keyword
String name = new String("Java String");
```

- In the above example, we have created a **string name** using the **new** keyword.
- Here, when we create a string object, the **String() constructor is invoked.**

Creating strings using the new keyword

- Example: Create Java Strings using the new keyword

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

    // create a string using new
    String name = new String("Java String");

    System.out.println(name);  // print Java String
  }
}
```

Create String using literals vs new keyword

- Now that we know how strings are created using string literals and the new keyword, let's see what is the major difference between them.
- In Java, the JVM maintains a **string pool to store all of its strings inside the memory.** The string pool helps in reusing the strings.

Create String using literals vs new keyword

- While creating strings using string literals,

```
String example = "Java";
```

- Here, we are directly providing the value of the string (Java). Hence, the compiler first checks the string pool to see if the string already exists.
- If the string already exists, the new string is not created. Instead, the new reference, example points to the already existed string (Java).
- If the string doesn't exist, the new string () Java is created.

Create String using literals vs new keyword

- While creating strings using the new keyword,

```
String example = new String("Java");
```

- Here, the value of the string is not directly provided. Hence, a new "Java" string is created even though "Java" is already present inside the memory pool.

Use the split Method to Parse a String in Java

- The **split() method of the String class works by splitting the source string keeping the original string unmodified, and returns an array of substrings** of the original string. This method has two variants.
- The **split(String regex) method takes a regular expression of type string as an argument** and splits the string around the regular expression's matches. If the regular expression fails to match any part of the original string, it returns an array with one element: the source string.
- The **split(String regex, int limit) method works the same but takes limit, which means how many strings to be returned.** If the limit is negative, the returned array can contain as many substrings as possible when the limit is 0. The array would contain all substrings, excluding the trailing empty strings.

Use the split Method to Parse a String in Java

- Output :

```
Output1 : March
Output2 : 705
Output2 : 103
Output2 : 102
Output2 : 456-123 : 112 _343-1 789----
```

```
public class StringTest {
    public static void main(String args []){
        String source1 = "March032021";
        String [] returnedArray1 = source1.split("\\d+");
        for(String str1 :  returnedArray1){
            System.out.println(" Output1 : "+str1);
        }

        String source2 = "950-003-123-900-456 : 11 _343-1 789----";
        String [] returnedArray2 = source2.split("-",4);
        for(String str2 :  returnedArray2){
            System.out.println(" Output2 : "+str2);
        }
    }
}
```

Methods of Java String

- Besides those mentioned above, there are various string methods present in Java.

Methods	Description
contains()	checks whether the string contains a substring
substring()	returns the substring of the string
join()	join the given strings using the delimiter
replace()	replaces the specified old character with the specified new character
replaceAll()	replaces all substrings matching the regex pattern
replaceFirst()	replace the first matching substring
charAt()	returns the character present in the specified location
getBytes()	converts the string to an array of bytes
indexOf()	returns the position of the specified character in the string
compareTo()	compares two strings in the dictionary order
compareToIgnoreCase()	compares two strings ignoring case differences
trim()	removes any leading and trailing whitespaces
format()	returns a formatted string

Methods	Description
split()	breaks the string into an array of strings
toLowerCase()	converts the string to lowercase
toUpperCase()	converts the string to uppercase
valueOf()	returns the string representation of the specified argument
toCharArray()	converts the string to a `char` array
matches()	checks whether the string matches the given regex
startsWith()	checks if the string begins with the given string
endsWith()	checks if the string ends with the given string
isEmpty()	checks whether a string is empty of not
intern()	returns the canonical representation of the string
contentEquals()	checks whether the string is equal to charSequence
hashCode()	returns a hash code for the string
subSequence()	returns a subsequence from the string

TIA-Academy

ASSIGNMENT 02

TIA-Academy

Java Access Modifiers

What are Access Modifiers?

- In Java, **access modifiers are used to set the accessibility** (visibility) of classes, interfaces, variables, methods, constructors, data members, and the setter methods. For example,

```
class Animal {
   public void method1() {...}

   private void method2() {...}
}
```

- In the above example, we have declared 2 methods: method1() and method2(). Here,
  - method1 is public - This means it can be accessed by other classes.
  - method2 is private - This means it can not be accessed by other classes.
- Note the keyword **public** and **private**. These are access modifiers in Java. They are also known as visibility modifiers.

Types of Access Modifier

- There are four access modifiers keywords in Java and they are:

Modifier	Description
Default	declarations are visible only within the package (package private)
Private	declarations are visible within the class only
Protected	declarations are visible within the package or all subclasses
Public	declarations are visible everywhere

Default Access Modifier

- If we do not explicitly specify any access modifier for classes, methods, variables, etc, then by default the default access modifier is considered. For example,

```
package defaultPackage;

class Logger {
    void message(){
        System.out.println("This is a message");
    }
}
```

- Here, the Logger class has the **default access modifier**. And the class is **visible to all the classes that belong to the defaultPackage package**. However, if we try to use the Logger class in another class outside of **defaultPackage**, we will get a compilation error.

Private Access Modifier

- When variables and methods are declared private, they cannot be accessed outside of the class. For example,

```
class Data {
    // private variable
    private String name;
}

public class Main {
    public static void main(String[] main){

        // create an object of Data
        Data d = new Data();

        // access private variable and field from another class
        d.name = "Programiz";
    }
}
```

Private Access Modifier

- In the above example, we have **declared a private variable named name**. When we run the program, we will get the following error:

```
Main.java:18: error: name has private access in Data
        d.name = "Programiz";
         ^
```

- The error is generated because **we are trying to access the private variable** of the Data class from the Main class.

Private Access Modifier

- You might be wondering what if we need to access those private variables. In this case, we can use the getters and setters method. For example,
- Output :

```
The name is Programiz
```

```
class Data {
    private String name;

    // getter method
    public String getName() {
        return this.name;
    }
    // setter method
    public void setName(String name) {
        this.name= name;
    }
}
public class Main {
    public static void main(String[] main){
        Data d = new Data();

        // access the private variable using the getter and setter
        d.setName("Programiz");
        System.out.println(d.getName());
    }
}
```

Private Access Modifier

- In the above example, we have a private variable named name. In order to access the variable from the outer class, we have used methods: **getName() and setName().** These methods are **called getter and setter in Java.**
- Here, we have used the **setter method (setName()) to assign value** to the variable and the **getter method (getName()) to access the variable**.
- We have used this keyword inside the **setName()** to refer to the variable of the class. To learn more on this keyword, visit Java this Keyword.
- Note: We cannot declare classes and interfaces private in Java. However, the nested classes can be declared private.

Protected Access Modifier

- When methods and data members are declared protected, we can access them within the same package as well as from subclasses. For example,
- Output :

```
I am an animal
```

```
class Animal {
    // protected method
    protected void display() {
        System.out.println("I am an animal");
    }
}

class Dog extends Animal {
    public static void main(String[] args) {

        // create an object of Dog class
        Dog dog = new Dog();
         // access protected method
        dog.display();
    }
}
```

Protected Access Modifier

- In the above example, we have a **protected method named display()** inside the Animal class. The Animal class is **inherited** by the Dog class.
- We then created an **object dog of the Dog class**. Using the object we tried to access the protected method of the parent class.
- Since **protected methods can be accessed from the child classes**, we are able **to access the method of Animal class from the Dog class**.
- Note: We cannot declare classes or interfaces protected in Java.

Public Access Modifier

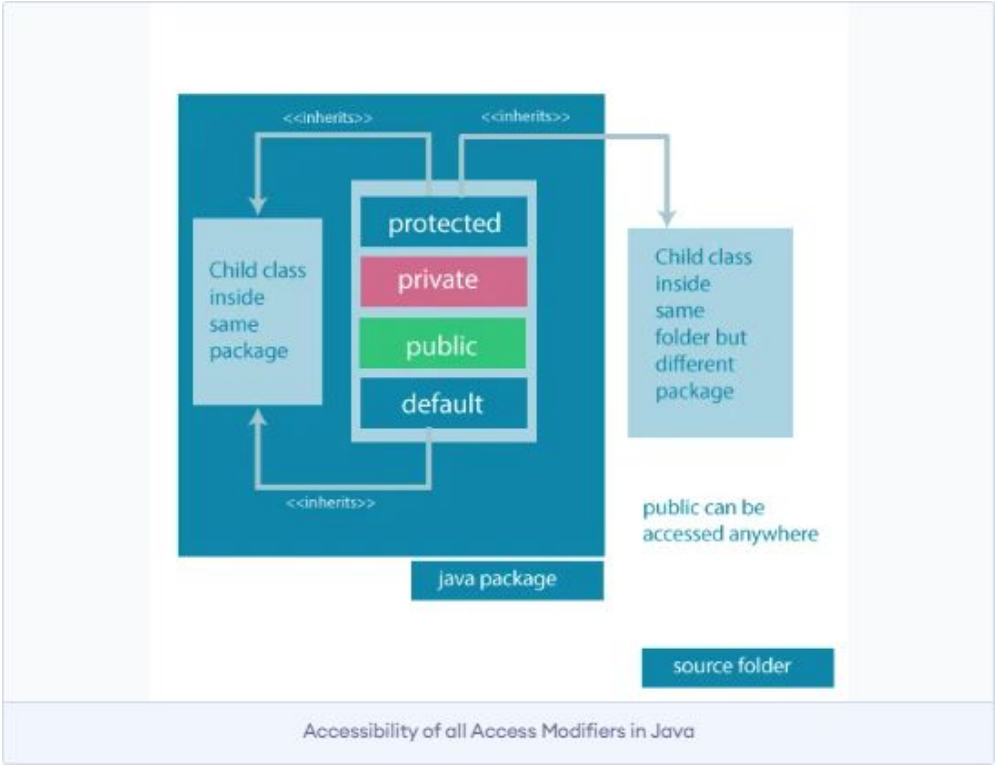
- When methods, variables, classes, and so on are declared public, then we can access them from anywhere.
- The public access modifier has no scope restriction. For example,
- Output :

```
I am an animal.
I have 4 legs.
```

```
// Animal.java file
// public class
public class Animal {
    // public variable
    public int legCount;

    // public method
    public void display() {
        System.out.println("I am an animal.");
        System.out.println("I have " + legCount + " legs.");
    }
}

// Main.java
public class Main {
    public static void main( String[] args ) {
        // accessing the public class
        Animal animal = new Animal();

        // accessing the public variable
        animal.legCount = 4;
        // accessing the public method
        animal.display();
    }
}
```

Access Modifiers Summarized in one figure

- Access modifiers are mainly used for encapsulation.
- It can help us to control what part of a program can access the members of a class. So that misuse of data can be prevented.



Accessibility of all Access Modifiers in Java

TIA-Academy

Java this Keyword

this Keyword

- In Java, this keyword is used to refer to the current object inside a method or a constructor. For example,
- Output :

```
this reference = Main@23fc625e
object reference = Main@23fc625e
```

```
class Main {
    int instVar;

    Main(int instVar){
        this.instVar = instVar;
        System.out.println("this reference = " + this);
    }

    public static void main(String[] args) {
        Main obj = new Main(8);
        System.out.println("object reference = " + obj);
    }
}
```

this Keyword

- In the above example, we created an **object named obj** of the class Main. We then **print the reference to the object obj** and this keyword of the class.
- Here, we can see that the reference of both obj and this is the same. It means this is nothing but the reference to the current object.

Use of this Keyword for Ambiguity Variable Names

- In Java, it is not allowed to declare two or more variables having the same name inside a scope (class scope or method scope). However, instance variables and parameters may have the same name. For example,

```
class MyClass {
    // instance variable
    int age;

    // parameter
    MyClass(int age){
        age = age;
    }
}
```

- In the above program, the instance variable and the parameter have the same name: age. Here, the Java compiler is confused due to name ambiguity. In such a situation, we use this keyword.

Use of this Keyword for Ambiguity Variable Names

- For example, first, let's see an example without using this keyword:
- Output :

```
obj.age = 0
```

```
class Main {

    int age;
    Main(int age){
        age = age;
    }

    public static void main(String[] args) {
        Main obj = new Main(8);
        System.out.println("obj.age = " + obj.age);
    }
}
```

Use of this Keyword for Ambiguity Variable Names

- In the above example, we have passed 8 as a value to the constructor. However, we are getting 0 as an output.
- This is because the Java compiler gets confused because of the ambiguity in names between instance the variable and the parameter.

Use of this Keyword for Ambiguity Variable Names

- Now, let's rewrite the above code using this keyword.
- Output :

```
obj.age = 8
```

```
class Main {

    int age;
    Main(int age){
        this.age = age;
    }

    public static void main(String[] args) {
        Main obj = new Main(8);
        System.out.println("obj.age = " + obj.age);
    }
}
```

Use of this Keyword for Ambiguity Variable Names

- Now, we are getting the expected output. It is because when the constructor is called, this inside the constructor is replaced by the object obj that has called the constructor.
- Hence the age variable is assigned value 8.

Use of this Keyword for Ambiguity Variable Names

- Also, if the name of the parameter and instance variable is different, the compiler automatically appends this keyword. For example, the code:

```
class Main {
    int age;

    Main(int i) {
        age = i;
    }
}
```

- is equivalent to:

```
class Main {
    int age;

    Main(int i) {
        this.age = i;
    }
}
```

Use of this with Getters and Setters

- Another common use of this keyword is in setters and getters methods of a class. For example:
- Output :

```
obj.age = Toshiba
```

- Here, we have used this keyword:
  - to assign value inside the setter method
  - to access value inside the getter method

```
class Main {
   String name;

   // setter method
   void setName( String name ) {
       this.name = name;
   }

   // getter method
   String getName(){
       return this.name;
   }

   public static void main( String[] args ) {
       Main obj = new Main();

       // calling the setter and the getter method
       obj.setName("Toshiba");
       System.out.println("obj.name: "+obj.getName());
   }
}
```

TIA-Academy

Java final Keyword

Java final keyword

- In Java, the **final keyword is used to denote constants**. It can be used with variables, methods, and classes.
- **Once any entity (variable, method or class) is declared final, it can be assigned only once.** That is,
  - the final variable cannot be reinitialized with another value
  - the final method cannot be overridden
  - the final class cannot be extended

Java final Variable

- In Java, we cannot change the value of a final variable. For example,

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

    // create a final variable
    final int AGE = 32;

    // try to change the final variable
    AGE = 45;
    System.out.println("Age: " + AGE);
  }
}
```

Java final Variable

- In the above program, we have created a final variable named age. And **we have tried to change the value** of the final variable.
- When we run the program, we will **get a compilation error** with the following message.

```
cannot assign a value to final variable AGE
    AGE = 45;
    ^
```

- Note: It is recommended to use uppercase to declare final variables in Java.

Java final Method

- In Java, the final method cannot be overridden by the child class. For example,

```
class FinalDemo {
    // create a final method
    public final void display() {
      System.out.println("This is a final method.");
    }
}

class Main extends FinalDemo {
  // try to override final method
  public final void display() {
    System.out.println("The final method is overridden.");
  }

  public static void main(String[] args) {
    Main obj = new Main();
    obj.display();
  }
}
```

Java final Method

- In the above example, we have created a **final method named display()** inside the FinalDemo class. Here, the **Main class inherits the FinalDemo class.**
- We have **tried to override the final method in the Main class**. When we run the program, we will get a compilation error with the following message.

```
display() in Main cannot override display() in FinalDemo
 public final void display() {
                   ^
 overridden method is final
```

Java final Class

- In Java, the final class cannot be inherited by another class. For example,

```
// create a final class
final class FinalClass {
  public void display() {
    System.out.println("This is a final method.");
  }
}

// try to extend the final class
class Main extends FinalClass {
  public  void display() {
    System.out.println("The final method is overridden.");
  }

  public static void main(String[] args) {
    Main obj = new Main();
    obj.display();
  }
}
```

Java final Class

- In the above example, we have created a final class named FinalClass. Here, we have tried to inherit the final class by the Main class.
- When we run the program, we will get a compilation error with the following message.

```
cannot inherit from final FinalClass
class Main extends FinalClass {
                   ^
```

TIA-Academy

Thank You