Java Basic for Tester

Java OOP (I)

Agenda

- *Java Class and Objects*
- *Java Methods*
- *Java Constructor*
- *Java Strings*
- *Java Access Modifiers*
- *Java this keyword*
- *Java final keyword*
- *Java Recursion*
- *Java instanceof Operator*

Java Class and Objects

Java is an object-oriented programming language. It is based on the concept of objects.

These objects share two characteristics:

- state (fields)
- behavior (methods)

Bicycle is an object
States: current gear, two wheels, number of gear, etc
Behavior: braking, accelerating, changing gears, etc

Before you create objects in Java, you need to define a class. A class is a blueprint for the object.

Java Class and Objects

Here's how we can define a class in Java:

```
class ClassName {
   // variables
   // methods
}
```

```java
package OOP1;

public class Lamp {
    // instance variable
    private boolean isOn;

    // method
    public void turnOn() {
        isOn = true;
    }

    // method
    public void turnOff() {
        isOn = false;
    }
}
```

Java Class and Objects

Java Objects: An object is called an instance of a class.

```
className object = new className();
```

Here, we are using the constructor className() to create the object. Constructors have the same name as the class and are similar to methods.

```
// l1 object
Lamp l1 = new Lamp();
// l2 object
Lamp l2 = new Lamp();
```

```
// access method turnOn()
l1.turnOn();
```

Java Class and Objects

Java Objects: An object is called an instance of a class.

```
class Lamp {

  ... .. ...
  void turnOn() {
    isOn = true;
  }

  ... .. ...
}

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

   ... .. ...
   l1.turnOn();
   ... .. ...

  }
}
```

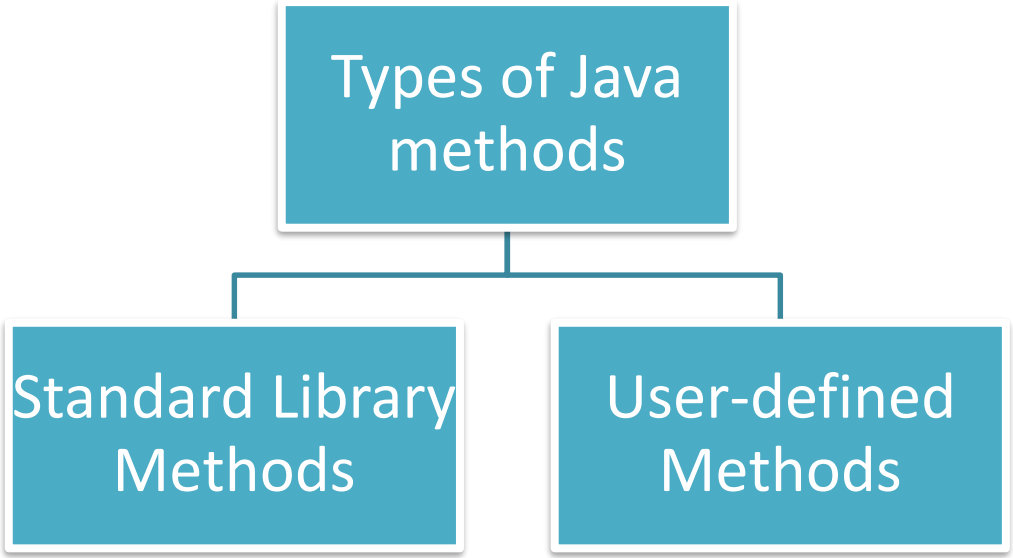
Java Class and Objects

```java
package OOP1;

public class Lamp {
    boolean isOn;

    void turnOn() {
        // initialize variable with value true
        isOn = true;
        System.out.println("Light on? " + isOn);

    }

    void turnOff() {
        // initialize variable with value false
        isOn = false;
        System.out.println("Light on? " + isOn);
    }
}

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

        // create objects l1 and l2
        Lamp l1 = new Lamp();
        Lamp l2 = new Lamp();

        // call methods turnOn() and turnOff()
        l1.turnOn();
        l2.turnOff();
    }
}
```

```
Run: Main
"C:\Program Files\Java\jdk1.8.0_21
Light on? true
Light on? false

Process finished with exit code 0
```

Java Methods

In object-oriented programming, the method is a jargon used for function. Methods are bound to a class and they define the behavior of a class.



Java Methods

```java
package OOP1;

public class JavaMethod {
    public static void main(String[] args) {
        System.out.println("About to encounter a method.");

        // method call
        myMethod();

        System.out.println("Method was executed successfully!");
    }

    // method definition
    private static void myMethod(){
        System.out.println("Printing from inside myMethod()!");
    }
}
```

Run: JavaMethod

```
"C:\Program Files\Java\jdk1.8.0_211\b
About to encounter a method.
Printing from inside myMethod()!
Method was executed successfully!

Process finished with exit code 0
```

Java Methods

```java
package OOP1;

public class SquareMain {
    public static void main(String[] args) {
        int result;

        // call the method and store returned value
        result = square();
        System.out.println("Squared value of 10 is: " + result);
    }

    public static int square() {
        // return statement
        return 10 * 10;
    }
}
```

Run: SquareMain

```
"C:\Program Files\Java\jdk1.8.0_21
Squared value of 10 is: 100

Process finished with exit code 0
```

Java Constructors

In Java, every class has its constructor that is invoked automatically when an object of the class is created. A constructor is similar to a method but in actual, it is not a method.

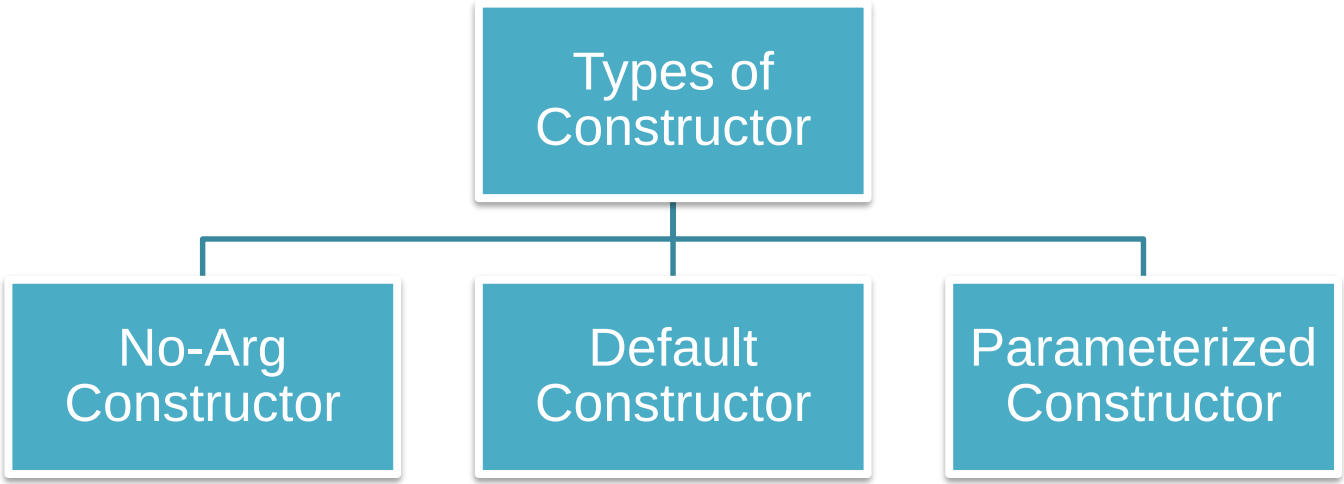
```
class Test {
    Test() {
        // constructor body
    }
}
```

Java Constructors

```java
package OOP1;

public class JavaConstructor {
    private int x;

    public JavaConstructor() {
        System.out.println("Constructor Called");
        x = 5;
    }

    public static void main(String[] args){
        // constructor is called while creating object
        JavaConstructor obj = new JavaConstructor();
        System.out.println("Value of x = " + obj.x);
    }
}
```

```
Run: JavaConstructor
"C:\Program Files\Java\jdk1.8.0_21
Constructor Called
Value of x = 5

Process finished with exit code 0
```

Java Constructors



No-Arg Constructor

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-arg constructor.

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

No-Arg Constructor

```java
package OOP1;

public class Company {
    String domainName;

    // public constructor
    public Company(){
        domainName = "lamns.com";
    }
}

 class Main1 {

     public static void main(String[] args) {

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

```
Run:  Main1
"C:\Program Files\Java\jdk1.8.0_211
Domain name = lamns.com

Process finished with exit code 0
```

Default Constructor

If you do not create any constructors, the Java compiler will automatically create a no-argument constructor during run-time. This constructor is known as the default constructor. The default constructor initializes any uninitialized instance variables with default values.

Type	Default Value
boolean	False
byte	0
Int	0
double	0.0d

Default Constructor

```java
package OOP1;

public class DefaultConstructor {
    int a;
    boolean b;

    public static void main(String[] args) {

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

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

```
Run: DefaultConstructor
"C:\Program Files\Java\jdk1.8.0_211
a = 0
b = false

Process finished with exit code 0
```

Parameterized Constructor

Similar to methods, we can pass parameters to a constructor. Such constructors are known as a parameterized constructor.

```
private Constructor (arg1, arg2, ..., argn) {
    // constructor body
}
```

Parameterized Constructor

```java
package OOP1;

public class Vehicle {
    int wheels;

    // constructor accepting single value
    private Vehicle(int wheels){
        this.wheels = wheels;
        System.out.println(wheels + " wheeler vehicle created.");
    }

    public static void main(String[] args) {

        // calling the constructor by passing single value
        Vehicle v1 = new Vehicle( wheels: 2);
        Vehicle v2 = new Vehicle( wheels: 3);
        Vehicle v3 = new Vehicle( wheels: 4);
    }
}
```

```
Run:  Vehicle
"C:\Program Files\Java\jdk1.8.0_211\bin\ja
2 wheeler vehicle created.
3 wheeler vehicle created.
4 wheeler vehicle created.

Process finished with exit code 0
```

Java Strings

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**'.
Unlike other programming languages, strings in Java are not primitive types (like int, char, etc). Instead, all strings are objects of a predefined class named **String**.

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

Java Strings

Java String provides various methods that allow us to perform different string operations. Here are some of the commonly used string methods.

Methods	Description
concat()	joins the two strings together
equals()	compares the value of two strings
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

Java Strings

Java String provides various methods that allow us to perform different string operations. Here are some of the commonly used string methods.

Methods	Description
length()	returns the size of the specified string
replace()	replaces the specified old character with the specified new character
substring()	returns the substring of the string
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 data

Java Strings

```java
package OOP1;

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

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

        //checks the string length
        System.out.println("The length of the string: " + greet.length());
    }
}
```

```
Run:  StringLength
"C:\Program Files\Java\jdk1.8.0_211
The string is: Hello! World
The length of the string: 12

Process finished with exit code 0
```

Java Strings

```java
package OOP1;

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

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

        String name = "World";
        System.out.println("Second String: " + name);

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

Run: StringConcat

```
"C:\Program Files\Java\jdk1.8.0_21
First String: Hello!
Second String: World
Joined String: Hello! World

Process finished with exit code 0
```

Java Strings

Strings in Java are represented by **double-quotes**.

```
// create a string
String example = "This is a string";
```

Now if we want to include **double-quotes** in our string

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

Java Strings

To solve this issue, the escape character \ is used in Java. Now we can include **double-quotes** in the string as:

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

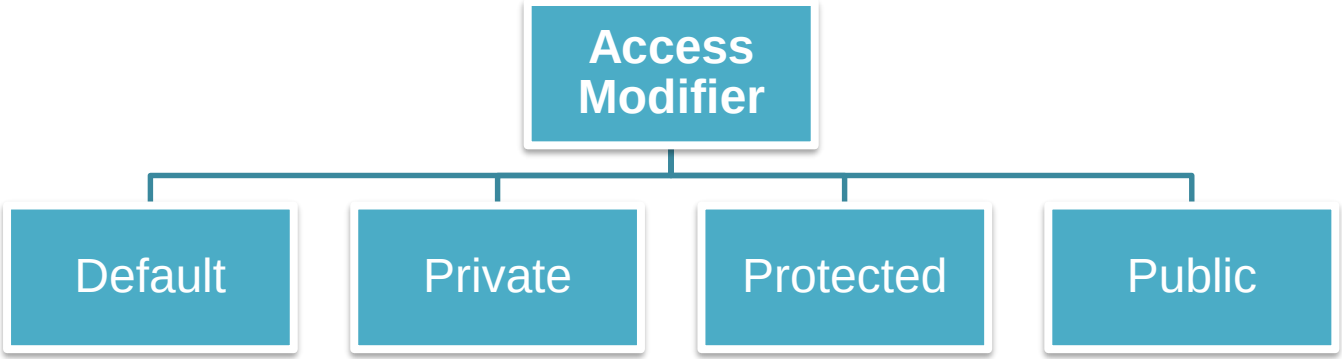
Java Access Modifiers

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

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

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

- **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.

Java Access Modifiers



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.

```java
package OOP1;

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

Private Access Modifier

```java
package OOP1;

public class Data {
    private String name;

    // getter method
    public String getName() {
        return this.name;
    }
    // setter method
    public void setName(String name) {
        this.name= name;
    }
}

class Main2 {
    public static void main(String[] main){
        Data d = new Data();

        // access the private variable using the getter and setter
        d.setName("LamNS3");
        System.out.println(d.getName());
    }
}
```

```
Run:  Main2
"C:\Program Files\Java\jdk1.8.0_21
LamNS3

Process finished with exit code 0
```

Protected Access Modifier

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

```java
package OOP1;

public 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();
    }
}
```

```
Run: Dog
"C:\Program Files\Java\jdk1.8.0_211\
I am an animal

Process finished with exit code 0
```

Java Access Modifiers

Public Access Modifier

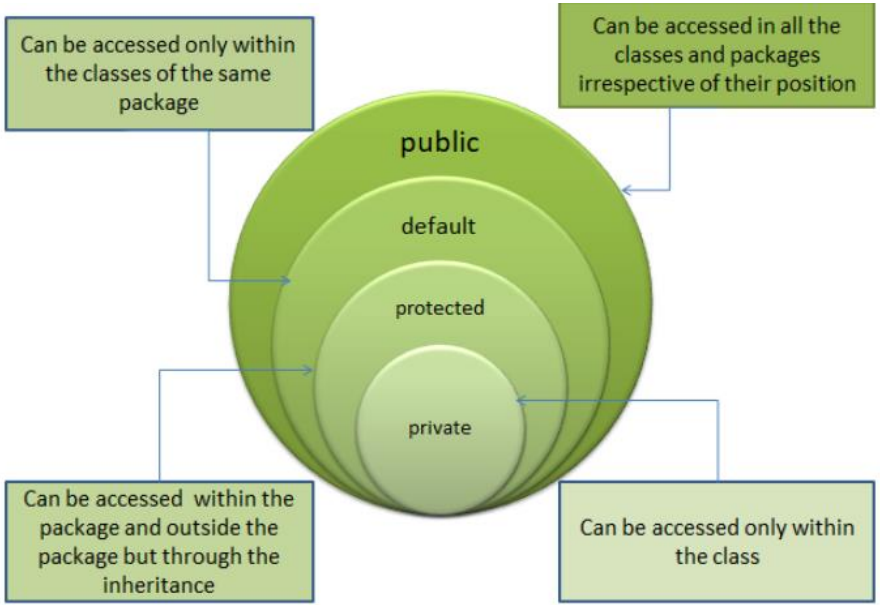
When methods, variables, classes, and so on are declared public, then we can access them from anywhere. The public access modifier has no scope restriction.

```
Run:  Main3
"C:\Program Files\Java\jdk1.8.0_21
I am an animal.
I have 4 legs.

Process finished with exit code 0
```

```java
package OOP1;

// 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
class Main3 {
    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();
    }
}
```

Java Access Modifiers

Access modifiers are mainly used for encapsulation. I 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.



Java this Keyword

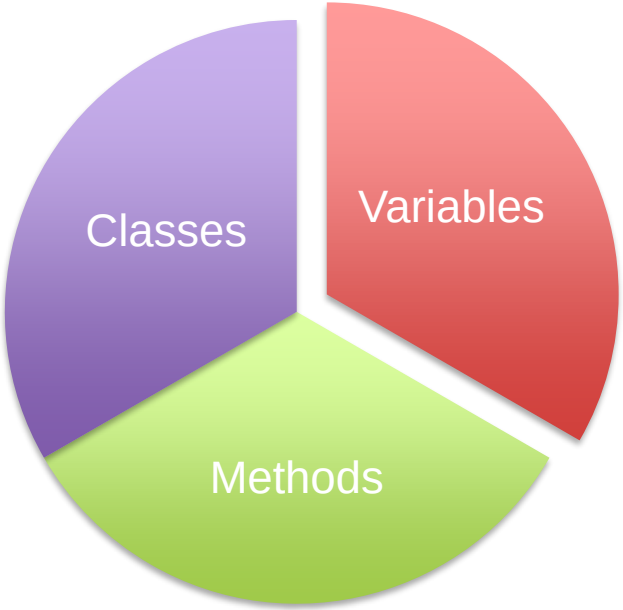
In Java, this keyword is used to refer to the current object inside a method or a constructor.

```java
package OOP1;

public class ThisKeyword {
    int instVar;

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

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

```
Run: ThisKeyword
"C:\Program Files\Java\jdk1.8.0_211\bin\java.e
this reference = OOP1.ThisKeyword@1b6d3586
object reference = OOP1.ThisKeyword@1b6d3586

Process finished with exit code 0
```

Java final keyword

In Java, the final keyword is used to denote constants.



Java final keyword

Java final Variable

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

```java
package OOP1;

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

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

        // try to change the final variable
        AGE = 45;
        S
    }
}
```

Cannot assign a value to final variable 'AGE'

Make 'AGE' not final Alt+Shift+Enter More actions... Alt+Enter

Java final keyword

Java final Method

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

```java
package OOP1;

public 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.printl
    }

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

'display()' cannot override 'display()' in 'OOP1.FinalDemo'; overridden method is final

Make 'FinalDemo.display' not final Alt+Shift+Enter More actions... Alt+Enter

Java final keyword

Java final Class

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

```java
package OOP1;

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

class Main extends FinalDemo {
    // try to override find
    public final void displ
        System.out.println(
    }
```

Cannot inherit from final 'OOP1.FinalDemo'

Make 'FinalDemo' not final Alt+Shift+Enter More actions... Alt+Enter

Java Recursion

In Java, a method that calls itself is known as a recursive method. And, this process is known as recursion.

```
public static void main(String[] args) {
    ... .. ...
    recurse()
    ... .. ...
}

static void recurse() {
    ... .. ...
    recurse()
    ... .. ...
}
```

Normal Method Call

Recursive Call

Java Recursion

```java
package OOP1;

public class Factorial {
    static int factorial( int n ) {
        if (n != 0)  // termination condition
            return n * factorial( n: n-1); // recursive call
        else
            return 1;
    }

    public static void main(String[] args) {
        int number = 4, result;
        result = factorial(number);
        System.out.println(number + " factorial = " + result);
    }
}
```

Run: Factorial

```
"C:\Program Files\Java\jdk1.8.0_211\
4 factorial = 24

Process finished with exit code 0
```

Java instanceof

In Java, the **instanceof** keyword is a binary operator. It is used to check whether an object is an instance of a particular class or not.

The syntax of the instanceof is:

```
result = objectName instanceof className;
```

Java instanceof

```java
package OOP1;

public class instanceofDemo {
    public static void main (String[] args) {
        String name = "LamNS3";
        Integer age = 22;

        System.out.println("Is name an instance of String: "+ (name instanceof String));
        System.out.println("Is age an instance of Integer: "+ (age instanceof Integer));
    }
}
```

```
Run:  instanceofDemo
"C:\Program Files\Java\jdk1.8.0_211\b
Is name an instance of String: true
Is age an instance of Integer: true

Process finished with exit code 0
```

Thank you