**About Me:**



**What is Java?**

* Java is a high-level, class-based, object-oriented programming language.
* It is a general-purpose programming language intended to let programmers write once, run anywhere (WORA)
* Java was originally developed by James Gosling at Sun Microsystems.
* It was released in May 1995.
* Now, it is owned by Oracle Corporation.

|  |  |
| --- | --- |
| **James Gosling** | **Java Logo** |



**History of Java:**

* Java programming language was initially called Oak after an oak tree that stood outside Gosling's office.
* Later the project went by the name Green.
* Finally renamed Java, from Java coffee, a type of coffee from Indonesia.
* Sun Microsystems released the first public implementation as Java 1.0 in 1996.
* Oracle Corporation's acquisition of Sun Microsystems happened 2009–10.
* On April 2, 2010, James Gosling resigned from Oracle.Now, he is working in Amazon Web Services.



**Java Version History**

|  |  |  |
| --- | --- | --- |
| **Version** | **Type** | **Release date** |
| **JDK 1.0** |  | **23 January 1996** |
| **JDK 1.1** |  | **18 February 1997** |
| **J2SE 1.2** |  | **4 December 1998** |
| **J2SE 1.3** |  | **8 May 2000** |
| **J2SE 1.4** |  | **13 February 2002** |
| **J2SE 5.0 (1.5)** |  | **30 September 2004** |
| **Java SE 6 (1.6)** |  | **11 December 2006** |
| **Java SE 7 (1.7)** |  | **28 July 2011** |
| **Java SE 8 (1.8)** | **LTS** | **18 March 2014** |
| **Java SE 9 (1.9)** |  | **21 September 2017** |
| **Java SE 10 (1.10)** |  | **20 March 2018** |
| **Java SE 11** | **LTS** | **25 September 2018** |
| **Java SE 12** |  | **19 March 2019** |
| **Java SE 13** |  | **17 September 2019** |
| **Java SE 14** |  | **17 March 2020** |
| **Java SE 15** |  | **16 September 2020** |
| **Java SE 16** |  | **16 March 2021** |
| **Java SE 17** | **LTS** | **14 September 2021** |
| **Java SE 18** |  | **22 March 2022** |
| **Java SE 19** |  | **20 September 2022** |
| **Java SE 20** |  | **21 March 2023** |
| **Java SE 21** | **LTS** | **19 September 2023** |
| **Java SE 22** |  | **19 March 2024** |
| **Java SE 23** |  | **17 September 2024** |
| **Java SE 24** |  | **March 2025** |
| **Java SE 25** | **LTS** | **September 2025** |

**Class #2**

---------

**Agenda**

-----------

print "My First Java Program" -> using java program

**4 steps**

----------

1- Write the Java Program

-> notepad++ (start -> search for notepad++)

-> language -> j -> java

2- Save the Java Program

-> file -> save as -> <class\_name>.java

3- Compile the Java Program

-> cmd -> in the file location

-> javac <class\_name>.java

-> <class\_name>.class file gets created

4- Run the Java Program

-> java <class\_name>

-> on comd prompt

**ProgramCount#1**

**class MyFirstJavaProgram**

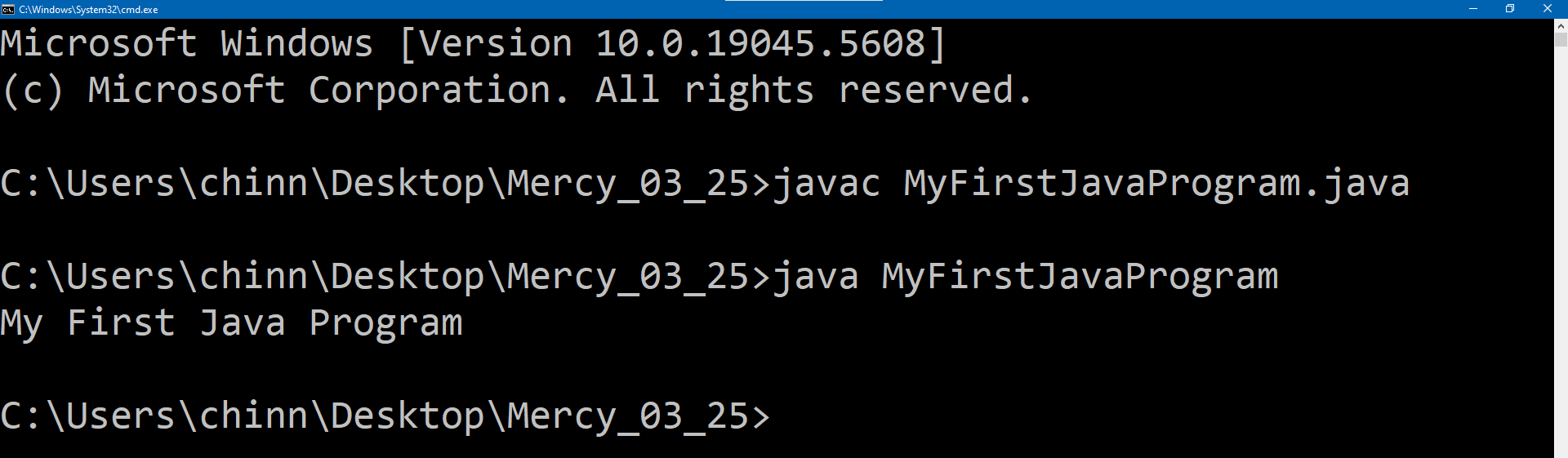
**{**

**public static void main(String[] args){**

**System.out.println("My First Java Program");**

**}**

**}**



**Class #3**

----------

Agenda

---------

**Printing various literals**

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

**what is a literal?**

-> in java, literal is a fixed value / a constant, that we can directly use in code.

-> in java, there is no restriction on how many literals you can use in a class

1- Printing various literals

a- String literals -> value written in double quotes ("")

b- integer literals -> number without decimal point -> 25, 55, 105

c- float literals -> number with decimal point ending with f -> 25.5f, 55.7f, 105.0f

d- char literals -> single value written in single quotes ('') -> a, y, @

e- boolean literals -> true / false

2- printing combination of literals

a- arithmatic expressions -> (+,-,\*,/,%)

b- boolean expressions -> expression evaluating to true / false

c- arithmatic expressions + String

**ProgramCount#2**

**class MyProgram1**

**{**

**public static void main(String[] args){**

**//printing String literals**

**System.out.println("Printing String literal 1");**

**System.out.println("Printing String literal 2");**

**System.out.println("Printing String literal 3");**

**System.out.println("Printing String literal 4");**

**}**

**}**

**Output:**

**----------**

Printing String literal 1

Printing String literal 2

Printing String literal 3

Printing String literal 4

**ProgramCount#3**

**class MyProgram2**

**{**

**public static void main(String[] args){**

**//printing integer literals**

**System.out.println(25);**

**System.out.println(55);**

**System.out.println(105);**

**}**

**}**

**Output:**

**----------**

25

55

105

**ProgramCount#4**

**class MyProgram3**

**{**

**public static void main(String[] args){**

**//printing floating point literals**

**System.out.println(25.5f);**

**System.out.println(55.7f);**

**System.out.println(105.0f);**

**}**

**}**

**Output:**

**----------**

25.5

55.7

105.0

**ProgramCount#5**

**class MyProgram4**

**{**

**public static void main(String[] args){**

**//printing char literals**

**System.out.println('a');**

**System.out.println('y');**

**System.out.println('@');**

**}**

**}**

**Output:**

**----------**

a

y

@

**ProgramCount#6**

**class MyProgram5**

**{**

**public static void main(String[] args){**

**//printing boolean literals**

**System.out.println(true);**

**System.out.println(false);**

**}**

**}**

**Output:**

**----------**

true

false

**ProgramCount#7**

**class MyProgram6 {**

**public static void main(String[] var0) {**

**System.out.println(15);**

**System.out.println(5);**

**System.out.println(50);**

**System.out.println(2);**

**System.out.println(0);**

**}**

**}**

**Output:**

**----------**

15

5

50

2

0

**ProgramCount#8**

**class MyProgram7**

**{**

**public static void main(String[] args){**

**//printing boolean expressions**

**System.out.println(100==100);//true**

**System.out.println(100==150);//false**

**System.out.println((20+80)==100);//true**

**System.out.println((20+80)==(150-40));//false**

**}**

**}**

**Output:**

**----------**

true

false

true

false

**ProgramCount#9**

**class MyProgram8**

**{**

**public static void main(String[] args){**

**//printing arithmatic expressions + String**

**//print result = 100**

**System.out.println("result = 100");**

**System.out.println("result = "+100);**

**System.out.println("result = "+(150+30));**

**}**

**}**

**Output:**

**----------**

result = 100

result = 100

result = 180

**Class #4**

-----------

Agenda

---------

1- **Java Data Types**

a- primitive data types

1- byte

2- short

3- int ->generally, we use int in place of byte / short

4- long

5- float

6- double

7- boolean

8- char

byte < short < int < long

float < double

b- non-pirmitive data types

-> these are derived from primitive data types

-> String, ArrayList, HashMap, Employee etc.

2- Java Variables

->water bottle example

->variable acts as storage to the actual value

byte b; // declaring variable

b=10; // assigning value

short s = 300; // single line declaring and assigning

syntax

--------

<data\_type> <variable\_name> --> example : byte b;

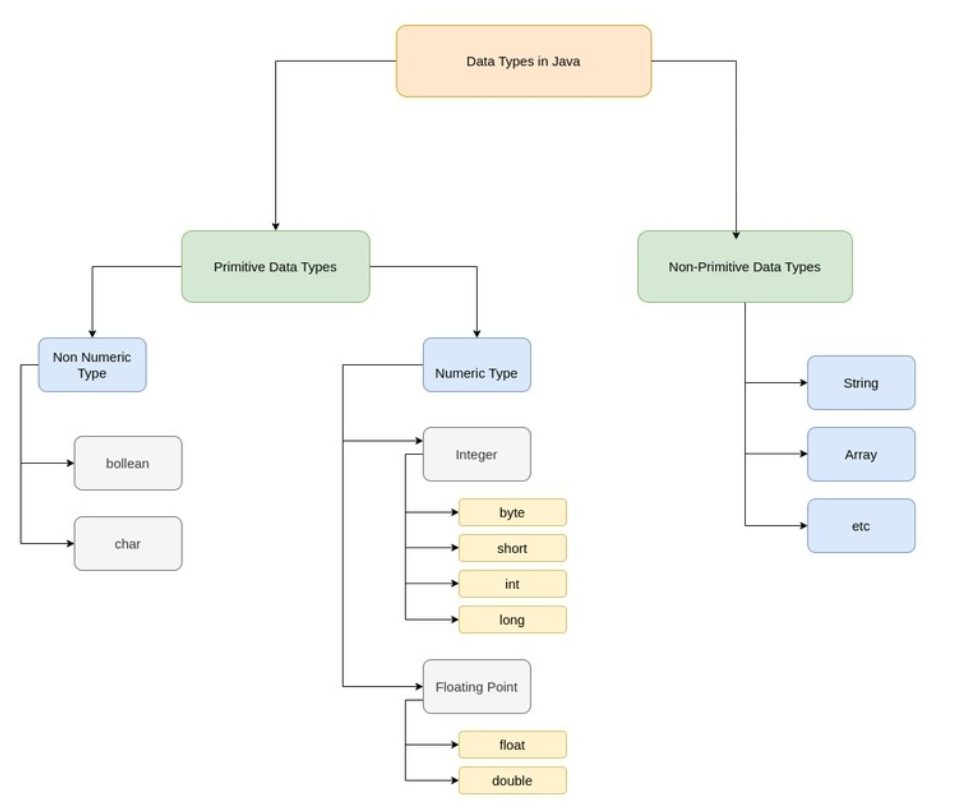
<data\_type> <variable\_name> = <value> --> short s = 300;

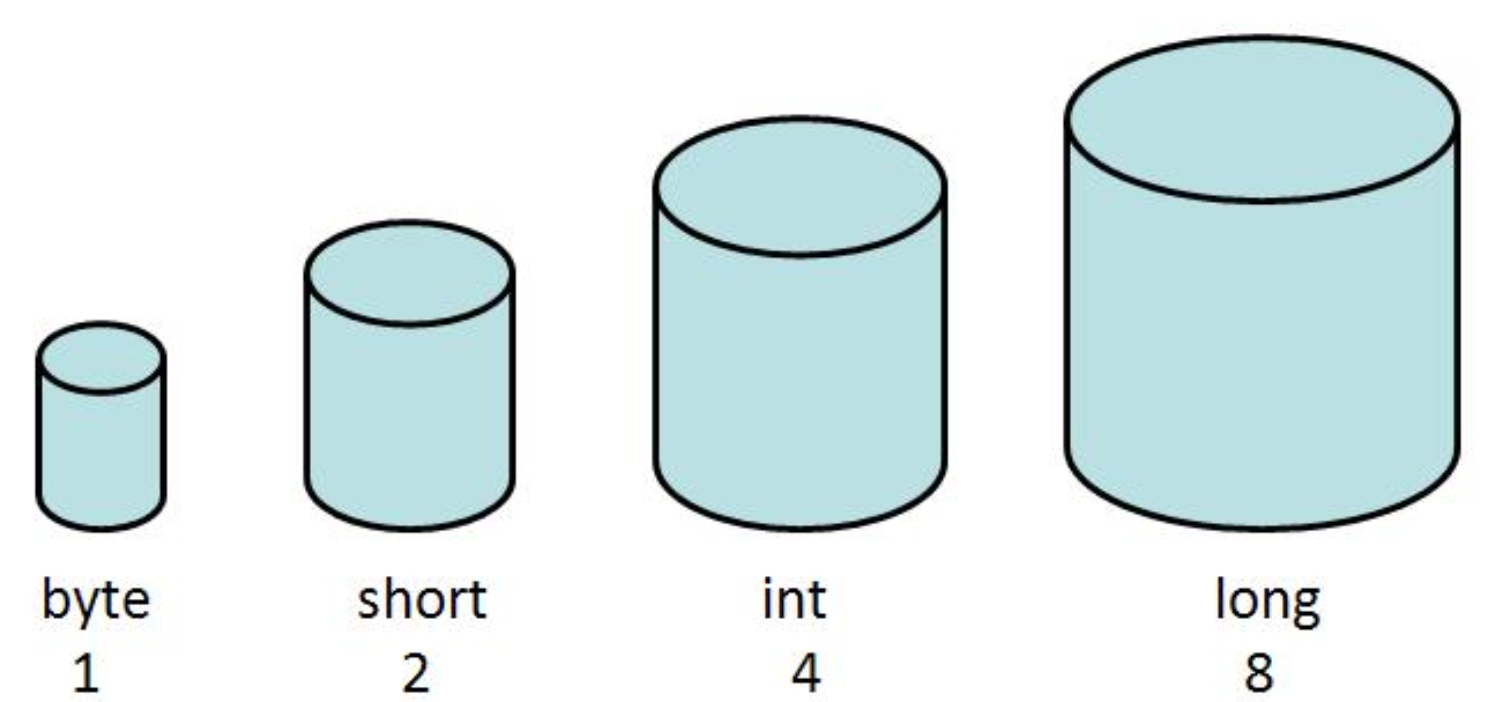
important points

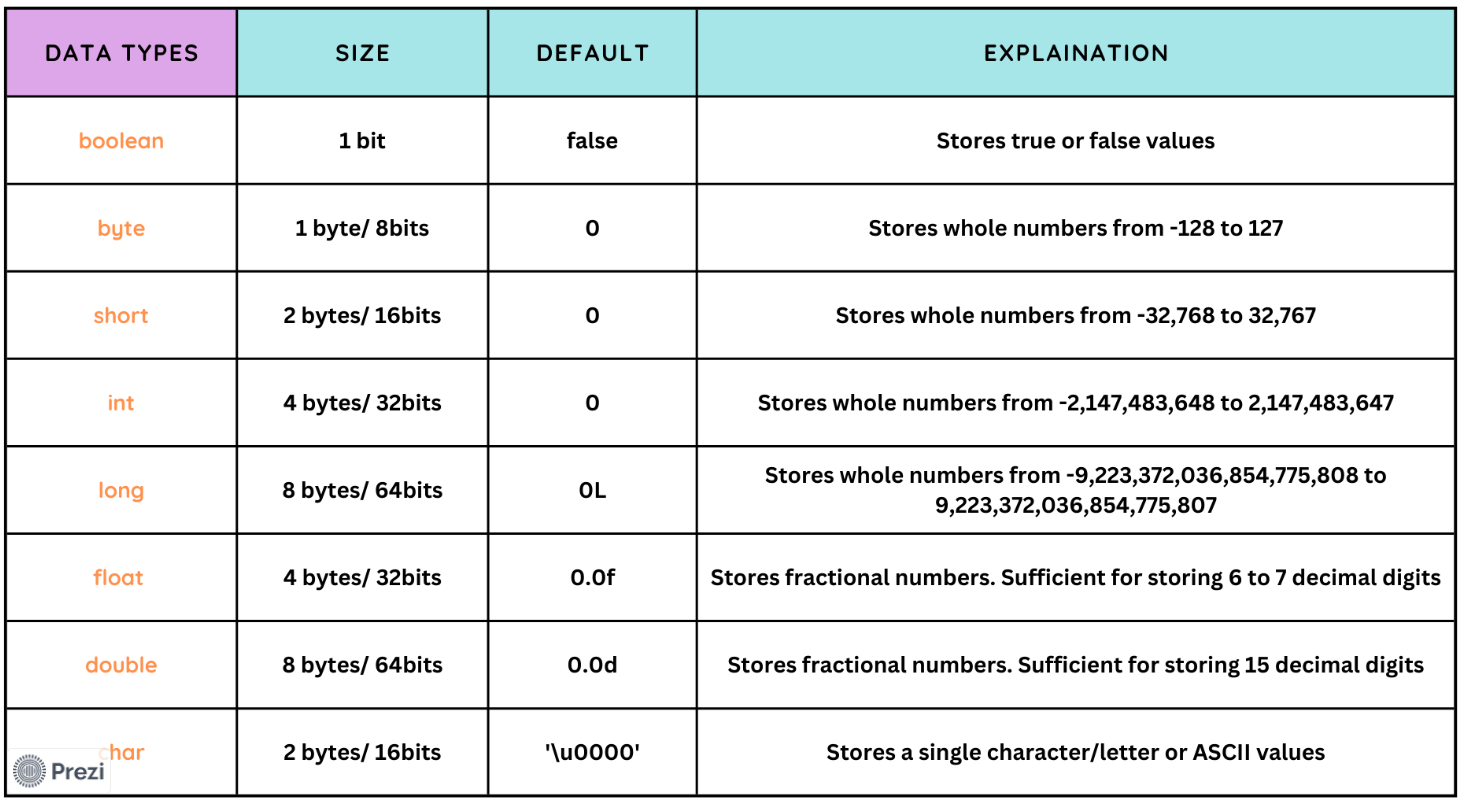
----------------------

1- We can re-use variables in code any number of times.

2- We can re-assign a variable in code any number of times.







**ProgramCount#10**

**class Variable1**

**{**

**public static void main(String[] args){**

**byte b;// b is a variable of type byte**

**b = 10;**

**System.out.println(b);**

**}**

**}**

**ProgramCount#11**

**class Variable2**

**{**

**public static void main(String[] args){**

**// Whole numbers**

**byte b; // declaring variable**

**b=10;   // assigning value**

**short s = 300; // single line declaring and assigning**

**int i = 1000;**

**long l = 2500;**

**System.out.println("b="+b);**

**System.out.println("s="+s);**

**System.out.println("i="+i);**

**System.out.println("l="+l);**

**// Decimal numbers**

**float f = 10.5f;**

**double d = 20.56d;**

**System.out.println("f="+f);**

**System.out.println("d="+d);**

**// boolean**

**boolean bool = true;**

**System.out.println("bool="+bool);**

**// char**

**char c = 'x';**

**System.out.println("c="+c);**

**}**

**}**

**ProgramCount#12**

**class Variable3**

**{**

**public static void main(String[] args){**

**int x = 50;**

**int y = x+20;**

**int z = x+y;**

**int m = x+y+z;**

**System.out.println("x="+x);**

**System.out.println("y="+y);**

**System.out.println("z="+z);**

**System.out.println("m="+m);**

**}**

**}**

**ProgramCount#13**

**class Variable4**

**{**

**public static void main(String[] args){**

**int x = 200;**

**System.out.println("x="+x);**

**x = 400;**

**System.out.println("x="+x);**

**}**

**}**

**Class #5**

**----------**

Agenda

----------

**Variable Usage and Rules**

======================

1- Variable naming rules

a- Variable name should follow camel case (start with lower case and followed by capital letter at start of each word)

-> employee name --> employeeName

-> student address --> studentAddress

-> person date of birth --> personDateOfBirth

b- variable names can contain

-> letters, numbers and underscore (\_)

-> can start with letter or underscore and not with number

-> employee1 -> valid

-> 1employee -> invalid (cannot start with number)

-> \_employee1 -> valid

-> employee\_1 -> valid

2- Some important points about variables

a- any variable declared inside a method is a local variable.

b- variable should be declared before it is used.

c- local variable cannot be used before it is initialized.

d- more than 1 local vriable can be used in a method.

e- we can assing one variable to another.

f- variable can be declared only once

**ProgramCount#14**

**class Variable5**

**{**

**public static void main(String[] args){**

**//Variable name should follow camel case**

**String employeeName = "Mercy";**

**// variable names can contain letters, numbers and underscore (\_)**

**String employee\_address\_1 = "mercy address";**

**// variable names cannot start with number**

**String 1\_employee\_address = "1 mercy address";**

**System.out.println("employeeName="+employeeName);**

**System.out.println("employee\_address\_1="+employee\_address\_1);**

**System.out.println("1\_employee\_address="+1\_employee\_address);**

**}**

**}**

**ProgramCount#15**

**class Variable6**

**{**

**public static void main(String[] args){**

**// local variable**

**String studentName = "Joseph";**

**System.out.println("studentName="+studentName);**

**// variable should be declared before it is used.**

**// System.out.println("studentAddress="+studentAddress);**

**// String studentAddress = "Joseph Address";**

**// local variable cannot be used before it is initialized.**

**// String studentDob;**

**// System.out.println("studentDob="+studentDob);**

**// we can assing one variable to another.**

**String studentName1 = "John";**

**System.out.println("studentName1="+studentName1);**

**studentName1 = studentName;**

**System.out.println("studentName1="+studentName1);**

**String studentName2 = "Adam";**

**System.out.println("studentName2="+studentName2);**

**String studentName2;**

**}**

**}**

**Class #6**

**-----------**

Agenda

---------

Internal working of java program & revision till date

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

1- programer writes a .java file

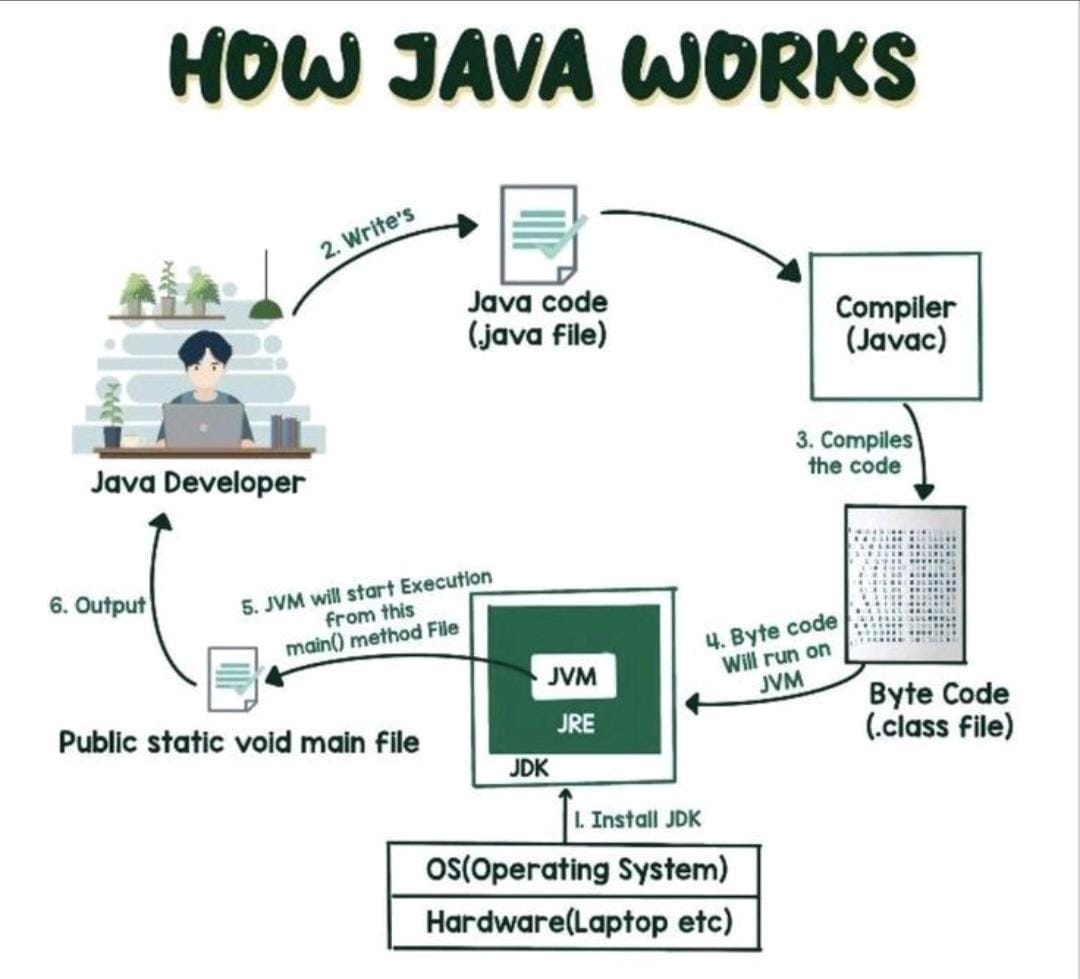
2- when we compile a .java file -> we get a .class file

a- programer cannot understand .class file

3- .class file is understood by JVM (JVM is inside JRE. JRE is inside JDK. JDK is installed in our laptop)

4- JVM will identify main method and starts execution from there.

5- if there is System.out.println() -> we can see the output on the console.



**ProgramCount#16**

**class MyJavaProgram**

**{**

**public static void main(String[] args){**

**System.out.println("Hello World!");**

**}**

**}**

**Class #7**

**----------**

Agenda

----------

**Methods in java**

**----------------------**

1- What is a method? Why do we need methods?

2- static method delcaration.

3- static method usage.

public static void main(String[] args){

// method body

}

**What is a method?**

* In Java, a method is a block of code that performs a specific task.
* It is defined within a class.
* It can be called or invoked to execute that task.

**Why do we need method?**

* Code Reusability: Write once, use multiple times; reduces redundancy.
* Modularity: Break down programs into smaller, manageable parts.
* Maintainability: Easier to update and fix specific parts.
* Readability: Descriptive method names improve understanding.

Syntax – static method

<access\_modifier> static <return\_type> <method\_name> <args>

{

// body

}

Public static void process(int id)

{

// code to process

}

Syntax – instance / non-static method

<access\_modifier> <return\_type> <method\_name> <args>

{

// body

}

Public static void process(int id)

{

// code to process

}

**Cafe (Class)**

* Collection of coffee machines (methods) for various drinks (tasks).

**Coffee Machine (Method)**

* Device to make a specific type of coffee (perform a task).

**Button on Coffee Machine (Method Name)**

* Indicates the type of coffee (task) to be made.

**Ingredients (Parameters)**

* Inputs needed to make the coffee (perform the task).

**Brewing Process (Method Body)**

* Steps to make the coffee (perform the task).

**Coffee Drink (Return Type)**

* The output of the coffee machine (method).



**Some important points about methods**

**-----------------------------------------------------**

1- if static keyword is present, then it is a static method.

2- if static keyword is not present, then it is non-static method or instance method.

3- static -> it belongs to class -> black board

4- non-static -> it belongs to object or instance -> book, pen, shoe etc.

5- every method should have return type. If you do not want to return anything, then use void.

6- every method can have zero or more arguements.

7- we should call the method to be executed

8- inside the method, we can have code to call other methods also.

9- 0 or more lines can be present in a method.

10 - we can call static method in 2 ways:

a- directly call using <method\_name>

b- call using <class\_name>.<method\_name>

z

**ProgramCount#17**

**class MyClass1**

**{**

**public static void main(String[] args){**

**System.out.println("Main method starts");**

**// directly calling init() method**

**init();**

**System.out.println("============");**

**// calling process() method**

**process();**

**System.out.println("============");**

**// calling destroy() using class name**

**int destroyResult = MyClass1.destroy();**

**System.out.println("return from destroy method :: "+destroyResult);**

**System.out.println("============");**

**System.out.println("Main method ends");**

**}**

**public static void init(){**

**// logic to initialize db connections**

**System.out.println("db connections initialized successfully!");**

**}**

**public static void process(){**

**// logic for the actual task**

**System.out.println("process method starts");**

**System.out.println("process method ends");**

**}**

**public static int destroy(){**

**// logic to destroy db connections**

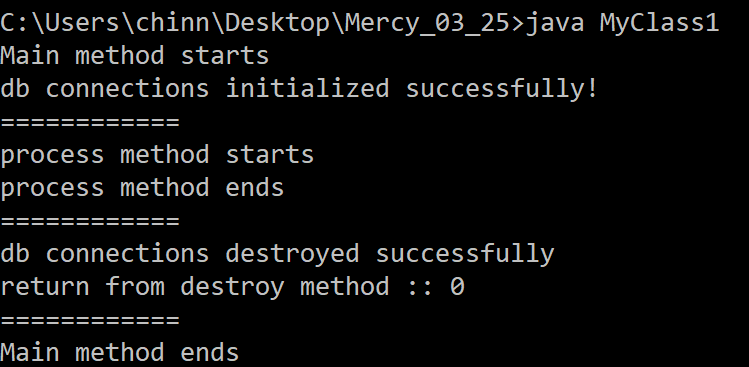
**// if successful -> return 0; else return 1;**

**System.out.println("db connections destroyed successfully");**

**return 0;**

**}**

**}**

****

Class #8

----------

Agenda

---------

Static Variables in java

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

static is applicable to variables and methods

static -> it belongs to class -> black board

types of memory

==============

1- Stack Memory - execution purpose - Ex. -> RAM

2. Heap Memory - storage - Ex. -> Hard Disk

Loading Operations

================

1. Whenever we are running a particular class, the class is loaded into stack memory.

2. When a particular class is loading into stack memory, all static variables are loaded into stack memory.

3. When a particular class is loading into stack memory, all static variables are loaded into stack memory with default values.

a- byte - 0

b- short - 0

c- int - 0

d- long - 0

e- float - 0.0

f- double - 0.0

e- boolean - false

g- char - ''

4. Once static variables are loaded with default values, then re-initialization happens from top to bottom.

5. When a particular class is loading into stack memory, all static methods are loaded into stack memory.

6. JVM will start executing main()

Static Members(variables & methods)

========================

1- class members

2- global members

3- universal members

**class StaticVariable1**

**{**

**// static variable**

**static int count;**

**// static method**

**public static void main(String[] args){**

**System.out.println("Welcome");**

**System.out.println("count = "+count);**

**}**

**}**

**class StaticVariable2**

**{**

**// primitive data types**

**static byte b;**

**static short s;**

**static int i;**

**static long l;**

**static float f;**

**static double d;**

**static boolean bool;**

**static char c;**

**public static void main(String[] args){**

**System.out.println("b="+b);**

**System.out.println("s="+s);**

**System.out.println("i="+i);**

**System.out.println("l="+l);**

**System.out.println("f="+f);**

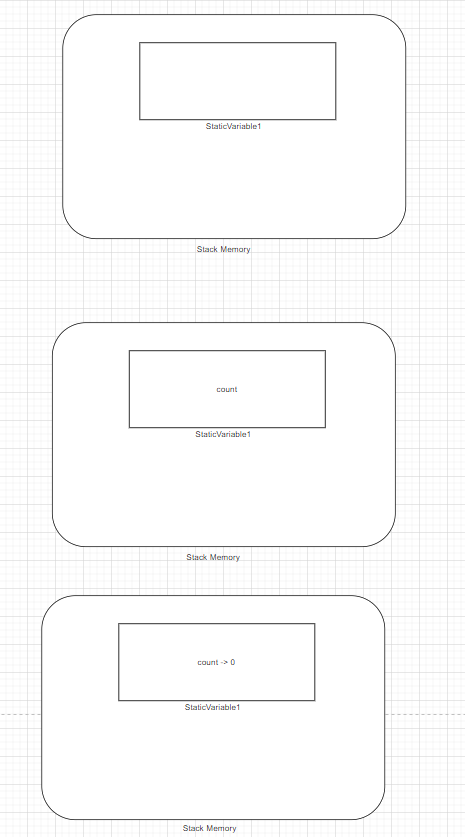
**System.out.println("d="+d);**

**System.out.println("bool="+bool);**

**System.out.println("c="+c);**

**}**

**}**

****

**class StaticVariable3**

**{**

**// static variable**

**static int count;**

**static int countNew = 100;**

**// static method**

**public static void main(String[] args){**

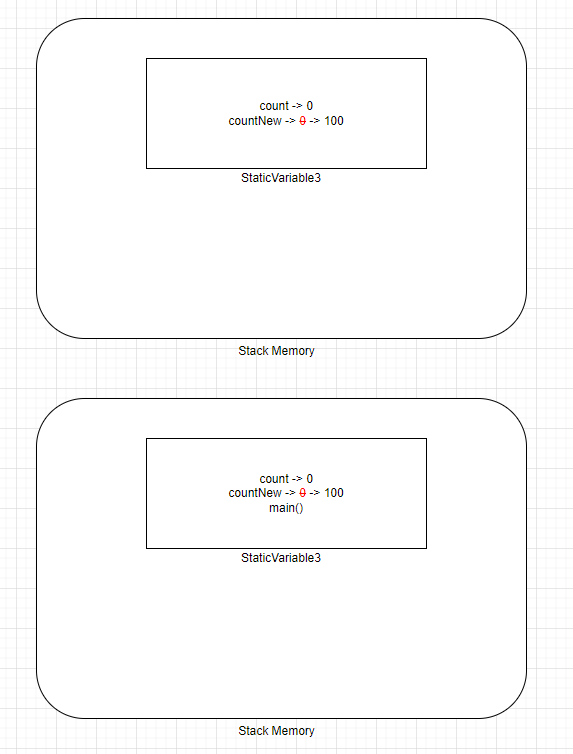
**System.out.println("Welcome");**

**System.out.println("count = "+count);**

**System.out.println("countNew = "+countNew);**

**}**

**}**

****

**class StaticVariable4**

**{**

**// static variable**

**static int count = 100;**

**// static method**

**public static void main(String[] args){**

**System.out.println("before calling process () - count :: "+count);**

**process();**

**System.out.println("after calling process () - count :: "+count);**

**}**

**// static method**

**public static void process(){**

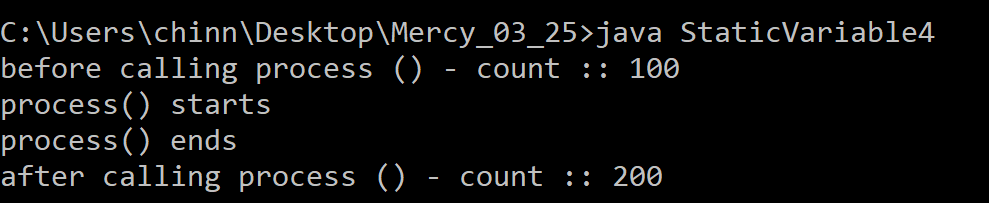
**System.out.println("process() starts");**

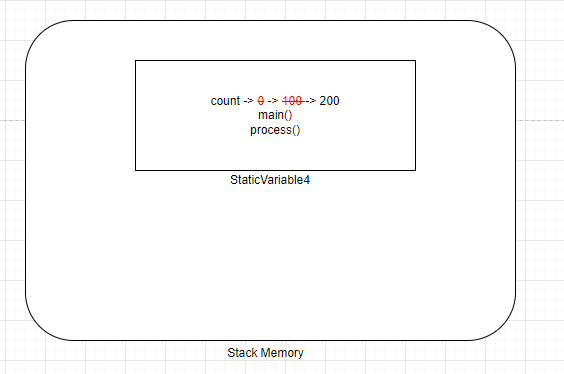
**count = 200;**

**System.out.println("process() ends");**

**}**

**}**

****



Class #9

----------

Agenda

--------

Static Initialization Block (SIB)

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

{

// block

}

static -> it belongs to class -> black board

// static block

static

{

// block

// contains initialization logic

}

1- Static Blocks are used to initialize globla variables (static variables).

2- Static Blocks are executed when class is loaded into stack memory.

3- SIBs are executed only once.

4- We can have multiple SIBs in a class.

5- SIBs are executed from top to bottom.

6- SIBs can be placed anywhere in class.(recommendation is to place SIBs in the begining of class)

7- SIBs are executed before main()

8- JVM will start executing main()

**class SIB1**

**{**

**// static block**

**static**

**{**

**System.out.println("from SIB 1");**

**}**

**// static block**

**static**

**{**

**System.out.println("from SIB 2");**

**}**

**public static void main(String[] args){**

**System.out.println("from main() method");**

**}**

**}**

**class SIB2**

**{**

**static int age = 10;**

**// static block**

**static**

**{**

**System.out.println("from SIB");**

**System.out.println("age = "+age);**

**age = 20;**

**}**

**public static void main(String[] args){**

**System.out.println("from main()");**

**System.out.println("age = "+age);**

**}**

**}**

**class SIB3**

**{**

**static int age = initAge();**

**// static block**

**static**

**{**

**System.out.println("SIB start");**

**main(null);**

**System.out.println("SIB end");**

**}**

**// static method**

**static int initAge(){**

**System.out.println("from initAge()");**

**System.out.println("age = "+age);**

**return 20;**

**}**

**public static void main(String[] args){**

**System.out.println("from main()");**

**System.out.println("age = "+age);**

**}**

**}**

Class #10

----------

Agenda

--------

Multiple Classes part - 1

===============

1- purpose of multiple classes

2- how to declare multiple classes

3- how to access members of one class file from another class file.

members -> methods & variables

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

1- class can be a part of .java file.

2- .java file can have multiple

a. classes

b. interfaces

c. enums

d. abstract classes

e. annotations

3- individual .class files will be generated for each of the above , when we compile a .java file

4- we need to use a specific .class file to run and get it executed.

5- How to access static members?

a- directly using their name (member name)

b- by using class name -> Syntax: <class\_name>.<member\_name>

6- if we are accessing one static member that belongs to different class, then we can use

<class\_name>.<member\_name>

**class MercyClass1**

**{**

**public static void main(String[] args){**

**System.out.println("from MercyClass1 main()");**

**}**

**}**

**class MercyClass2**

**{**

**public static void main(String[] args){**

**System.out.println("from MercyClass2 main()");**

**}**

**}**

**class MercyClass2**

**{**

**public static void main(String[] args){**

**}**

**}**

**class MercyClass2New**

**{**

**}**

**interface MercyClass2Interface**

**{**

**}**

**enum MercyClass2Enum**

**{**

**}**

**abstract class MercyClass2AbstractClass**

**{**

**}**

**@interface MercyClass2Annotation**

**{**

**}**

**class MercyClass3**

**{**

**static int count = 10;**

**static int getCount(){**

**System.out.println("from MercyClass3 getCount()");**

**return 15;**

**}**

**public static void main(String[] args){**

**System.out.println("from MercyClass3 main()");**

**System.out.println("count = "+count);**

**System.out.println("getCount() = "+getCount());**

**System.out.println("===========");**

**System.out.println("value = "+MercyClass4.value);**

**System.out.println("getValue() = "+MercyClass4.getValue());**

**}**

**}**

**class MercyClass4**

**{**

**static int value = 20;**

**static int getValue(){**

**System.out.println("from MercyClass4 getValue()");**

**return 25;**

**}**

**public static void main(String[] args){**

**System.out.println("from MercyClass4 main()");**

**System.out.println("value = "+value);**

**System.out.println("getValue() = "+getValue());**

**System.out.println("===========");**

**System.out.println("value = "+MercyClass3.count);**

**System.out.println("getValue() = "+MercyClass3.getCount());**

**}**

**}**

===================

Class #11 #12 & #13 by Sashi Sir

===================

Class #14

-----------

Agenda

=====

IDE introduction (Intellij Idea)

===================

IDE -> Integrated Development Environment

community edition -> free edition

Jet Brains -> company owns Intellij Idea

file -> new project -> MercyCoreJava\_03\_2025

creating new java file -> file -> src -> rt click -> new java class

main -> suggession for main method -> to use it, do double click

sout -> suggession for System.out.println() -> to use it, do double click

**public class MyFirstJavaProgram {  
 public static void main(String[] args) {  
 System.*out*.println("My First Java Program Using IntelliJ");  
 System.*out*.println("My Second Java Program Using IntelliJ");  
 System.*out*.println("My Third Java Program Using IntelliJ");  
 System.*out*.println("My Fourth Java Program Using IntelliJ");  
 System.*out*.println("My Fifth Java Program Using IntelliJ");  
 }  
}**

### How to Download and Install IntelliJ IDEA Community Edition on Windows

Follow these steps to download and install IntelliJ IDEA Community Edition on your Windows PC:

#### ****Step 1: Download IntelliJ IDEA****

1. Open your browser and go to the official JetBrains website: <https://www.jetbrains.com/idea/download/>
2. Under the **Community** section, click on **Download** (the Community Edition is free and open-source).

#### ****Step 2: Run the Installer****

1. Once the download is complete, locate the downloaded .exe file in your Downloads folder.
2. Double-click the file to start the installation.
3. If prompted by User Account Control (UAC), click **Yes** to allow the installation.

#### ****Step 3: Install IntelliJ IDEA****

1. The setup wizard will open. Click on **Next**.
2. Choose the installation destination folder (default location is usually fine) and click **Next**.
3. In the next window, you can select the following options based on your preferences:
   * **Create Desktop Shortcut**: Check the box to create a shortcut on your desktop.
   * **Update PATH Variable**: Select this to add IntelliJ IDEA to the system PATH for command-line use.
   * **Associate with .java files**: If you're working with Java files, check this box.
4. Click on **Next** and then **Install**.

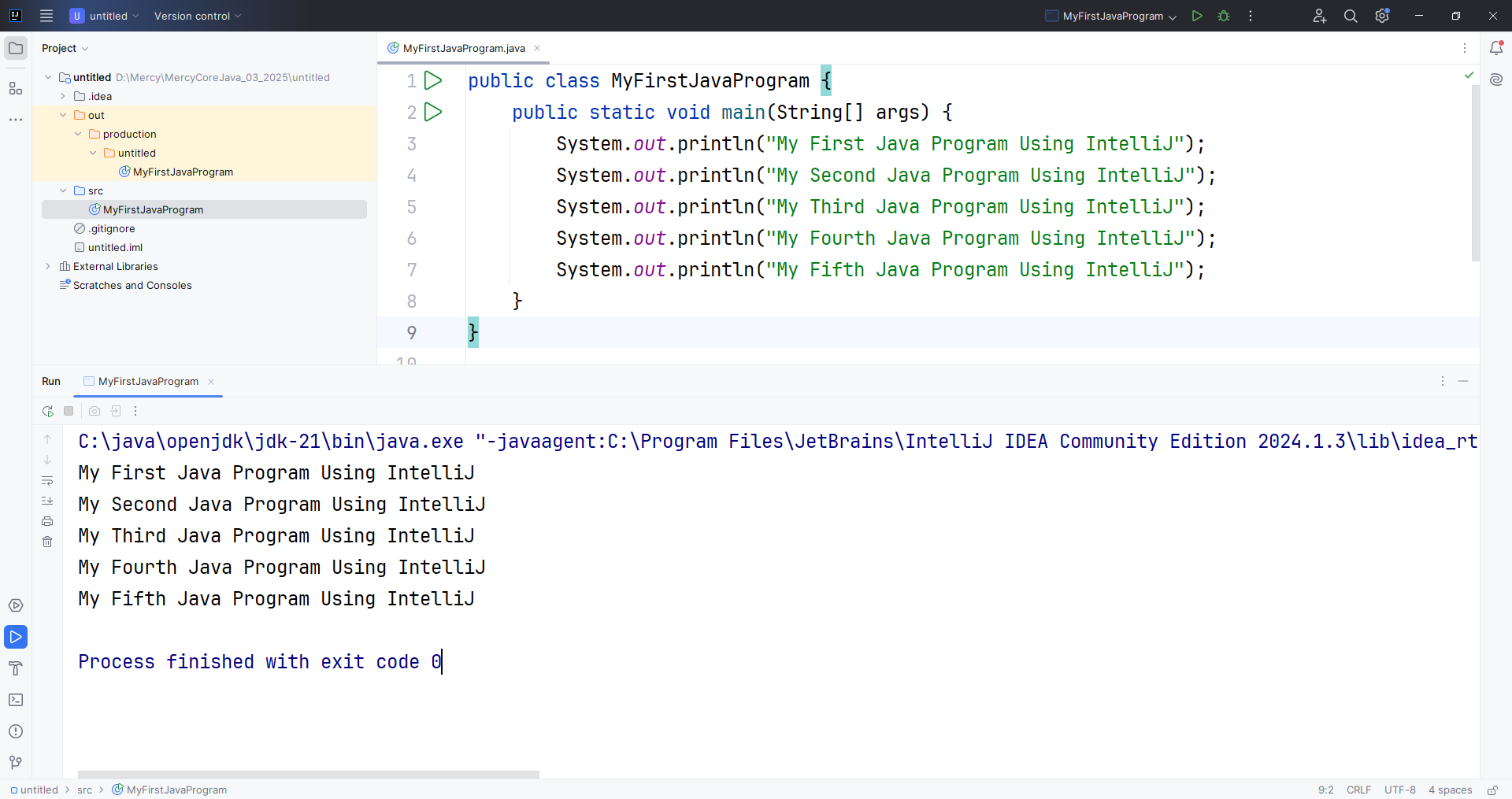
#### ****Step 4: Launch IntelliJ IDEA****

1. After the installation completes, check the box for **Run IntelliJ IDEA** and click **Finish**.
2. On the first launch, you may be asked to import settings. If you don't have any previous settings, select **Do not import settings** and click **OK**.
3. Choose the theme (Light or Dark) and complete the initial setup wizard.

#### ****Step 5: Install Java (if required)****

* If you haven't installed Java yet, download the latest version of the **Java Development Kit (JDK)** from <https://www.oracle.com/java/technologies/downloads/>.
* After installation, configure the JDK path in IntelliJ IDEA.

You are now ready to create your first Java project using IntelliJ IDEA Community Edition. Enjoy coding!



Class #15

-----------

Agenda

======

Multiple classes Part - 2

1- There should be only 1 public class in a java file. The name of the java file should be the same as public class name.

2- ctrl+y -> short cut for delete the line.

3- method calls should avoid unconditional call.

Exception in thread "main" java.lang.StackOverflowError

4- method calls should avoid unconditional recursion.

Exception in thread "main" java.lang.StackOverflowError

recursion -> method calling itself.

5- more than 1 static block can be present in a class

6- static blocks can be present in any place of the class. but execution happens from top to bottom.

7- SIBs are executed only once at the time of class loading.

**public class MercyClass4 {  
 static int *count* = 10;  
 static int getCount(){  
 System.*out*.println("from MercyClass4 getCount()");  
 return 15;  
 }  
 public static void main(String[] args) {  
 System.*out*.println("MercyClass4 main() method start");  
 System.*out*.println("count = "+*count*);  
 System.*out*.println("getCount() = "+*getCount*());  
 System.*out*.println("MercyClass4 main() method end");  
 System.*out*.println("=============================");  
 System.*out*.println("value = "+MercyClass5.*value*);  
 System.*out*.println("getValue() = "+MercyClass5.*getValue*());  
 }  
}  
  
class MercyClass5 {  
 static int *value* = 20;  
 static int getValue(){  
 System.*out*.println("from MercyClass5 getValue()");  
 return 25;  
 }  
 public static void main(String[] args) {  
 System.*out*.println("MercyClass5 main() method start");  
 System.*out*.println("MercyClass5 main() method end");  
 }  
}**

MercyClass4 main() method start

count = 10

from MercyClass4 getCount()

getCount() = 15

MercyClass4 main() method end

=============================

value = 20

from MercyClass5 getValue()

getValue() = 25

**public class MercyClass6 {  
 public static void main(String[] args) {  
 System.*out*.println("MercyClass6 main() start");  
 MercyClass7.*main*(null);  
 System.*out*.println("MercyClass6 main() end");  
 }  
}  
  
class MercyClass7{  
 public static void main(String[] args) {  
 System.*out*.println("MercyClass7 main() start");  
 MercyClass6.*main*(null);  
 System.*out*.println("MercyClass7 main() end");  
 }  
}**

MercyClass6 main() start

MercyClass7 main() start

Exception in thread "main" java.lang.StackOverflowError

**public class MercyClass8 {  
 public static void main(String[] args) {  
 System.*out*.println("MercyClass8 main() start");  
 *main*(null);  
 System.*out*.println("MercyClass8 main() end");  
 }  
}**

MercyClass8 main() start

Exception in thread "main" java.lang.StackOverflowError

**public class MercyClass9 {  
 static {  
 System.*out*.println("MercyClass9 static block 1");  
 }  
 public static void main(String[] args) {  
 System.*out*.println("MercyClass9 main() start");  
 MercyClass10.*main*(null);  
 MercyClass10.*main*(null);  
 MercyClass10.*main*(null);  
 MercyClass10.*main*(null);  
 MercyClass10.*main*(null);  
 MercyClass10.*main*(null);  
 MercyClass10.*main*(null);  
 System.*out*.println("MercyClass9 main() end");  
 }  
 static {  
 System.*out*.println("MercyClass9 static block 2");  
 }  
}  
  
class MercyClass10{  
 static {  
 System.*out*.println("MercyClass10 static block 1");  
 }  
 public static void main(String[] args) {  
 System.*out*.println("MercyClass10 main() start");  
 System.*out*.println("MercyClass10 main() end");  
 }  
}**

MercyClass9 static block 1

MercyClass9 static block 2

MercyClass9 main() start

MercyClass10 static block 1

MercyClass10 main() start

MercyClass10 main() end

MercyClass10 main() start

MercyClass10 main() end

MercyClass10 main() start

MercyClass10 main() end

MercyClass10 main() start

MercyClass10 main() end

MercyClass10 main() start

MercyClass10 main() end

MercyClass10 main() start

MercyClass10 main() end

MercyClass10 main() start

MercyClass10 main() end

MercyClass10 main() start

MercyClass10 main() end

MercyClass10 main() start

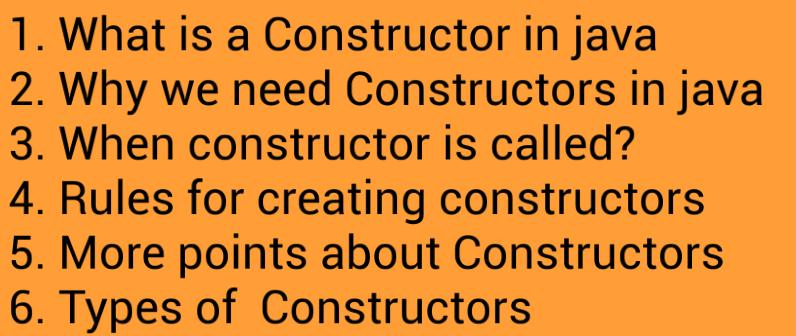
MercyClass10 main() end

MercyClass9 main() end

**Class #15**

**-----------**

**Constructors:**



**Why do we need constructors?**

1. Constructors are used to initialize instance variables.

2. In other words, constructors are used to initialize the object’s state.

3. The purpose of a constructor is to initialize the newly created object before it is used.

**When do we need constructors?**

1. Constructor is called when an object is instantiated ie created

2. In other words, when you use the new keyword while creating object, constructor is called

**Rules**

1. Constructor name should be the same name as its class name

2. A Constructor should not have any return type (not even void)

3. A Constructor cannot be abstract, static, final, and synchronized

**Important points**

1. Constructors can be declared anywhere inside the class

2. Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor.

**Types of constructors**

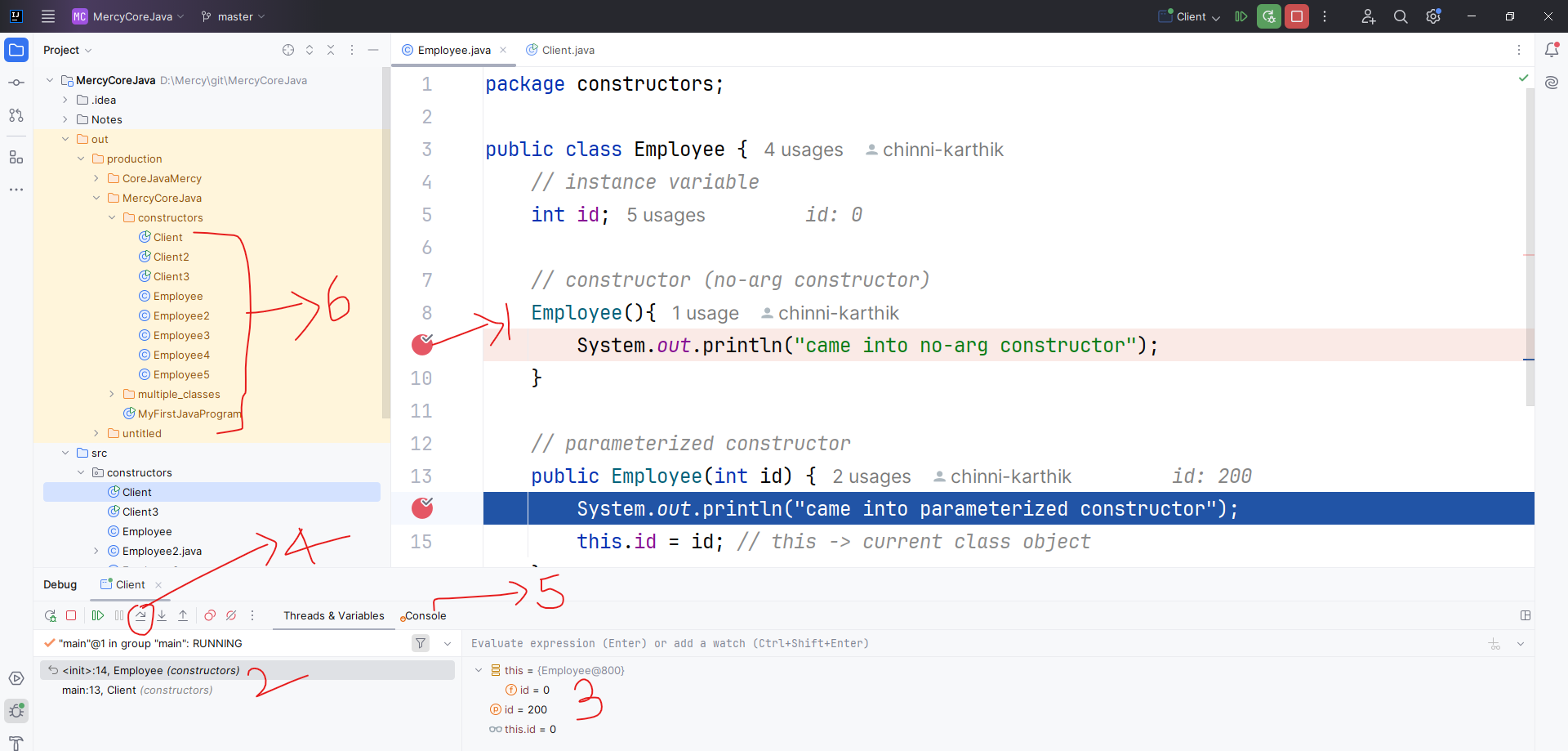
1. no-arg constructor

2. Parameterized constructor

3. default constructor

**public class Employee {  
 *// instance variable* int id;  
  
 *// constructor (no-arg constructor)* Employee(){  
 System.*out*.println("came into no-arg constructor");  
 }  
  
 *// parameterized constructor* public Employee(int id) {  
 System.*out*.println("came into parameterized constructor");  
 this.id = id; *// this -> current class object* }  
}**

**public class Client {  
 public static void main(String[] args) {  
 Employee e1 = new Employee();  
 System.*out*.println("e1.id = "+e1.id);  
  
 *// initializing object state* e1.id=100;  
 System.*out*.println("e1.id = "+e1.id);  
  
 *// initializing object state using constructor* Employee e2 = new Employee(200);  
 System.*out*.println("e2.id = "+e2.id);  
 }  
}**



* 1. Debug poing
  2. Stack trace
  3. Variable & threads
  4. Step Over (move to next line)
  5. Console (to see the sout statements)
  6. .class files generated

**public class Employee2 {  
 *// instance variable* int id;  
}  
  
class Client2{  
 public static void main(String[] args) {  
 Employee2 e2 = new Employee2();  
 System.*out*.println("e2.id = "+e2.id);  
 }  
}**

**public class Employee3 {  
 *// instance variable* int id;  
 String name;  
 String company;  
  
 public Employee3() {  
 }  
  
 public Employee3(int id) {  
 this.id = id;  
 }  
  
 public Employee3(String name) {  
 this.name = name;  
 }  
  
 public Employee3(int id, String name) {  
 this.id = id;  
 this.name = name;  
 }  
  
 public Employee3(int id, String name, String company) {  
 this.id = id;  
 this.name = name;  
 this.company = company;  
 }  
}**

**public class Client3 {  
 public static void main(String[] args) {  
 Employee3 e3 = new Employee3();  
 System.*out*.println("e3.id = "+e3.id);  
 System.*out*.println("e3.name = "+e3.name);  
 System.*out*.println("e3.company = "+e3.company);  
  
 System.*out*.println("=====================");  
  
 Employee3 e33 = new Employee3(300,"Mercy");  
 System.*out*.println("e33.id = "+e33.id);  
 System.*out*.println("e33.name = "+e33.name);  
 System.*out*.println("e33.company = "+e33.company);  
  
 System.*out*.println("=====================");  
  
 Employee3 e333 = new Employee3(300,"Mercy", null);  
 System.*out*.println("e333.id = "+e333.id);  
 System.*out*.println("e333.name = "+e333.name);  
 System.*out*.println("e333.company = "+e333.company);  
 }  
}**

e3.id = 0

e3.name = null

e3.company = null

=====================

e33.id = 300

e33.name = Mercy

e33.company = null

=====================

e333.id = 300

e333.name = Mercy

e333.company = null

**Class #16**

**Agenda**

1. What is constructor overloading?

2. Why we need constructor overloading?

3. Rules for constructor overloading?

**What is constructor overloading?**

Constructor overloading allows having more than one constructor inside one Class

**Why we need constructor overloading?**

If we want to have different ways of initializing an object using different number of parameters

**Rules for constructor overloading?**

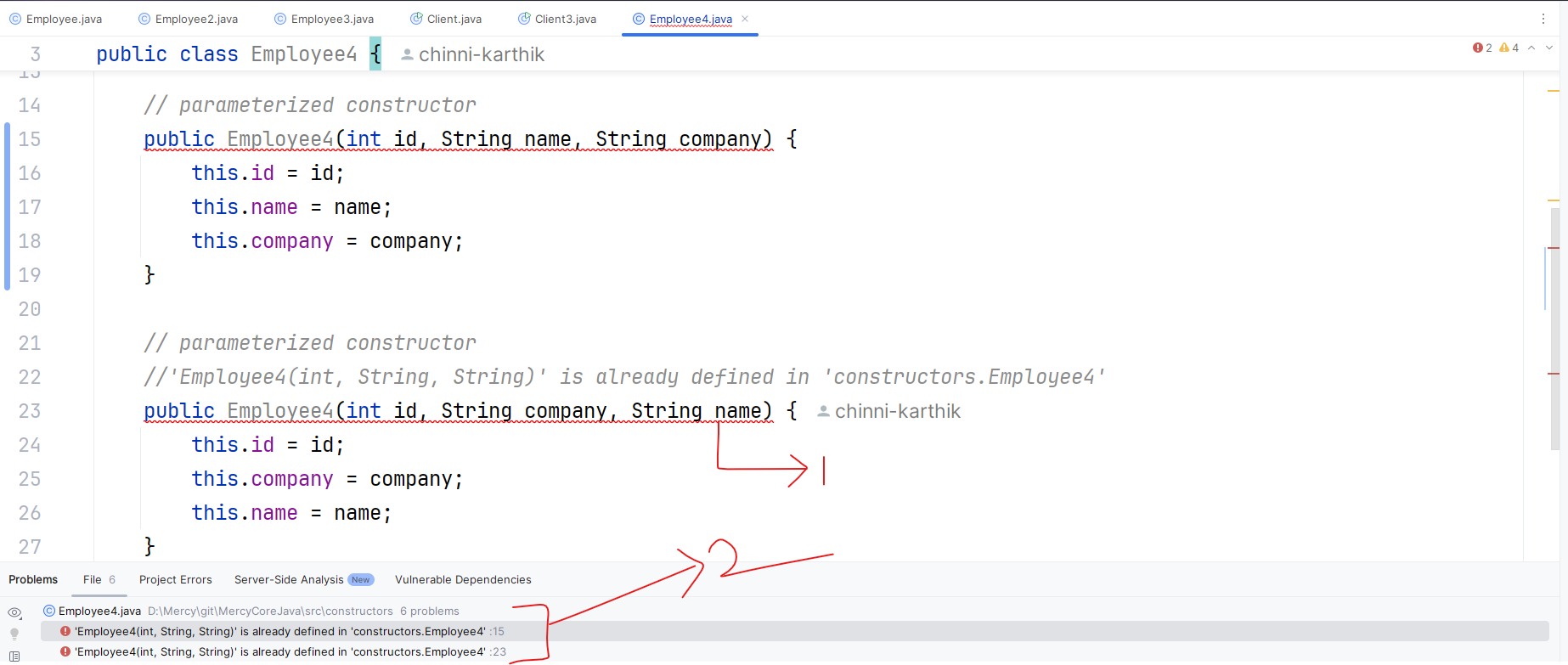
Different should be as like below,

1. Number of arguments

2. Type of arguments

3. Sequence of arguments

**public class Employee4 {  
 *// instance variables* int id;  
 String name;  
 String company;  
  
 *// zero-arg constructor* public Employee4() {  
 System.*out*.println("zero-arg constructor");  
 }  
  
 *// parameterized constructor* public Employee4(int id, String name, String company) {  
 this.id = id;  
 this.name = name;  
 this.company = company;  
 }  
  
 *// parameterized constructor  
 //'Employee4(int, String, String)' is already defined in 'constructors.Employee4'* public Employee4(int id, String company, String name) {  
 this.id = id;  
 this.company = company;  
 this.name = name;  
 }  
}**



1-> error indication by red line

2-> errors present in the present file

**public class Employee5 {  
 *// instance variables* int id;  
 String name;  
 float salary;  
  
 *// zero-arg constructor* public Employee5() {  
 System.*out*.println("zero-arg constructor");  
 }  
  
 *// parameterized constructor* public Employee5(int id, String name, int salary) {  
 this.id = id;  
 this.name = name;  
 this.salary = salary;  
 }  
  
 *// parameterized constructor  
 // 'Employee5(int, int, String)' is already defined in 'constructors.Employee5'  
 // you get above error if salary is int* public Employee5(int id, float salary, String name) {  
 this.id = id;  
 this.salary = salary;  
 this.name = name;  
 }  
  
 *// parameterized constructor* public Employee5(float salary, int id, String name) {  
 this.salary = salary;  
 this.id = id;  
 this.name = name;  
 }  
}**

**Class #17**

**Agenda**

**----------**

1. What is this() in Constructor ?

2. Why we need this() in Constructor ?

3. Some important points

**What is this() in constructor?**

1. this() can be used to invoke current class constructor.

2. this() can be used inside the constructor to call another overloaded constructor in the same Class.

3. It is called as Explicit Constructor Invocation.

4. This technique is called as constructor chaining

**Some important points**

this( ) keyword should be the first statement in Constructor.

**public class Employee6 {  
 *// instance variables* int empId;  
 String empName;  
  
 *// zero-arg constructor* public Employee6() {  
 System.*out*.println("inside zero-arg constructor");  
 }  
  
 *// parameterized constructor* public Employee6(int empId) {  
 this.empId = empId;  
 System.*out*.println("inside parameterized constructor 1");  
 }**

***// parameterized constructor* public Employee6(String empName) {  
 this.empName = empName;  
 System.*out*.println("inside parameterized constructor 2");  
 }  
  
 *// parameterized constructor* public Employee6(int empId, String empName) {  
 this.empId = empId;  
 this.empName = empName;  
 System.*out*.println("inside parameterized constructor 3");  
 }  
}**

**public class Client6 {  
 public static void main(String[] args) {  
 System.*out*.println("main() starts");  
 Employee6 emp6 = new Employee6();  
 System.*out*.println("emp6.empId = "+emp6.empId);  
 System.*out*.println("emp6.empName = "+emp6.empName);  
  
 System.*out*.println("============================");  
  
 Employee6 emp66 = new Employee6(100,"Mercy");  
 System.*out*.println("emp66.empId = "+emp66.empId);  
 System.*out*.println("emp66.empName = "+emp66.empName);  
  
 System.*out*.println("main() ends");  
 }  
}**

**public class Employee7 {  
 *// instance variables* int empId;  
 String empName;  
  
 *// zero-arg constructor* public Employee7() {  
 *//Call to 'this()' must be first statement in constructor body  
 //System.out.println("inside zero-arg constructor");  
 //this calling statement* this(250,"John");  
 }  
  
 public Employee7(int empId, String empName) {  
 *//Recursive constructor invocation  
 //this();* this.empId = empId;  
 this.empName = empName;  
 }  
}**

**public class Client7 {  
 public static void main(String[] args) {  
 Employee7 emp7 = new Employee7();  
 System.*out*.println("emp7.empId : "+emp7.empId);  
 System.*out*.println("emp7.empName : "+emp7.empName);  
 }  
}**

**Class #18**

1. Static Initialization Blocks (SIBs)

2. Constructors

3. Constructors Overloading

4. this() calling statement

**public class Employee8 {  
 *// ctrl+alt+l -> this is for code formatting* static {  
 System.*out*.println("Employee8 SIB1");  
 }  
  
 static {  
 System.*out*.println("Employee8 SIB2");  
 }  
  
 *// instance variables* int empId;  
 String empName;  
  
 *// zero-arg constructor* public Employee8() {  
 this(100);  
 System.*out*.println("Employee8() zero-arg constructor");  
 }  
  
 *// parameterized constructor* public Employee8(int empId) {  
 this(empId, "Mercy");  
 this.empId = empId;  
 }  
  
 *// parameterized constructor* public Employee8(int empId, String empName) {  
 this.empId = empId;  
 this.empName = empName;  
 }  
}**

**public class Client8 {  
 static {  
 System.*out*.println("Client8 SIB1");  
 }  
 static {  
 System.*out*.println("Client8 SIB2");  
 }  
 public static void main(String[] args) {  
 Employee8 emp8 = new Employee8();  
 System.*out*.println("emp8.empId : "+emp8.empId);  
 System.*out*.println("emp8.empName : "+emp8.empName);  
 }  
}**

Client8 SIB1

Client8 SIB2

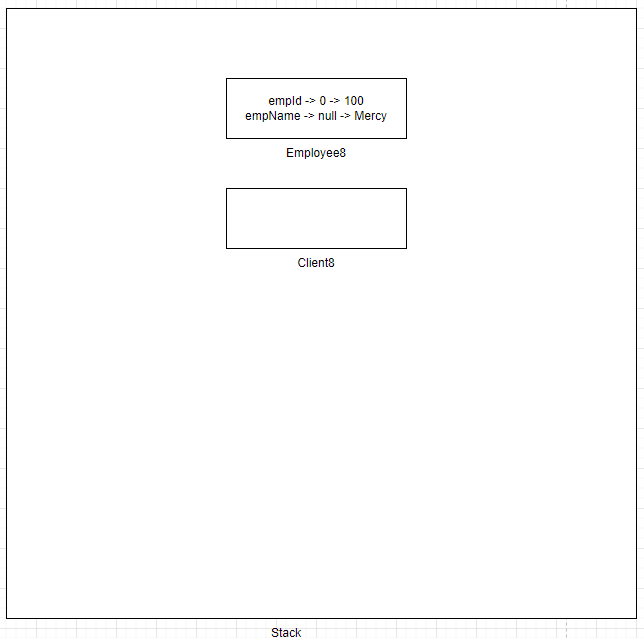
Employee8 SIB1

Employee8 SIB2

Employee8() zero-arg constructor

emp8.empId : 100

emp8.empName : Mercy



**Class #19**

**Inheritance**

**Agenda**

1. What is inheritance?
2. Why do we need inheritance? -> code reusability.
3. Benefits of inheritance.

**What is inheritance?**

Inheriting super class properties to sub class

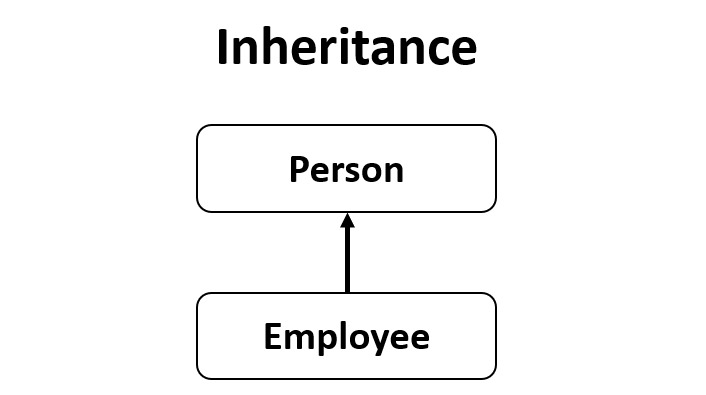
Reusability can be achieved

Inheritance is one of the important OOP concept

Inheriting attributes and methods from super class to sub class.

Only instance (non-static) attributes & methods can be inherited

Static attributes, static methods, constructors, static blocks, Instance initialization blocks cannot be inherited from super class





**public class Person {  
 *// instance variables* String name;  
 String dateOfBirth;  
 String gender;  
  
 public Person(String name, String dateOfBirth, String gender) {  
 this.name = name;  
 this.dateOfBirth = dateOfBirth;  
 this.gender = gender;  
 }  
  
 *// instance method* public int findAge() {  
 System.*out*.println("logic for findAge()");  
 return 0;  
 }  
  
 @Override  
 public String toString() {  
 return "Person{" +  
 "name='" + name + '\'' +  
 ", dateOfBirth='" + dateOfBirth + '\'' +  
 ", gender='" + gender + '\'' +  
 '}';  
 }  
}**

**public class Employee extends Person {  
 *// instance variables* int empId;  
 boolean isPermanent;  
  
 public Employee(String name, String dateOfBirth,**

**String gender, int empId, boolean isPermanent) {  
 super(name, dateOfBirth, gender);  
 this.empId = empId;  
 this.isPermanent = isPermanent;  
 }  
  
 *// instance method* public void findPfDetails() {  
 System.*out*.println("logic for findPfDetails()");  
 }  
  
 @Override  
 public String toString() {  
 return "Employee{" +  
 "empId=" + empId +  
 ", isPermanent=" + isPermanent +  
 "} " + super.toString();  
 }  
}**

**public class Client {  
 public static void main(String[] args) {  
 Person p = new Person("Mercy", "01-01-2020", "Female");  
 System.*out*.println(p);  
  
 Employee e = new Employee("Mercy","01-01-2020","Female", 1000, true);  
 System.*out*.println(e);  
 }  
}**

Output:

Person{name='Mercy', dateOfBirth='01-01-2020', gender='Female'}

Employee{empId=1000, isPermanent=true} Person{name='Mercy', dateOfBirth='01-01-2020', gender='Female'}

**Class #20**

**Inheritance - 2**

1. parent -> child => inheritance
2. grandparent -> parent -> child => **multi level inheritance**
3. When child class object is created (constructor is invoked), then parent class constructor is called (as part of constructor chaining.)
4. compiler adds super() or this() when programmer does not add them.
5. this() -> used to call current class constructor
6. super() -> used to call parent constructor
7. For all java classes, there is a parent class -> Object.java (Object class)
8. Below combinations are not allowed(this() and super() are not allowed at the same time in a constructor)

super();

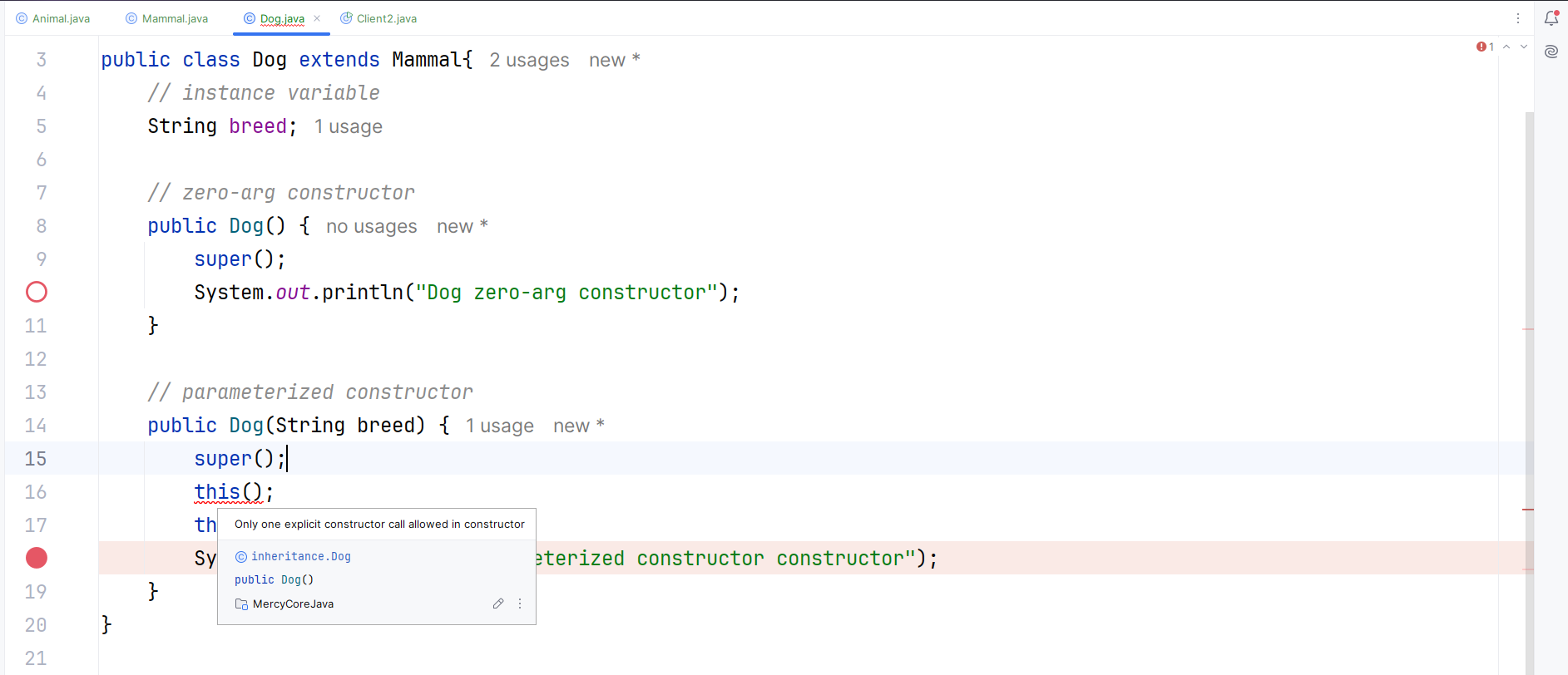
this();

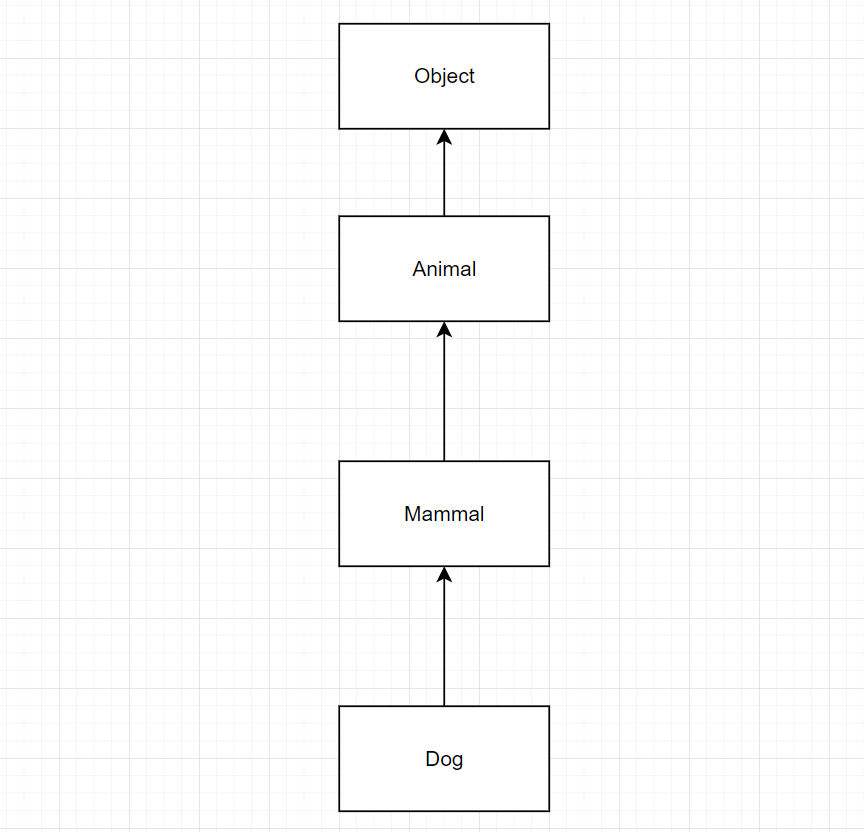
super();

super();

this();

this();





**public class Animal {  
  
 *// zero-arg constructor* public Animal() {  
 super();  
 System.*out*.println("Animal zero-arg constructor");  
 }  
}**

**public class Mammal extends Animal{  
  
 *// zero-arg constructor* public Mammal() {  
 super();  
 System.*out*.println("Mammal zero-arg constructor");  
 }  
}**

**public class Dog extends Mammal{  
 *// instance variable* String breed;  
  
 *// zero-arg constructor* public Dog() {  
 super();  
 System.*out*.println("Dog zero-arg constructor");  
 }  
  
 *// parameterized constructor* public Dog(String breed) {  
 this();  
 this.breed = breed;  
 System.*out*.println("Dog parameterized constructor constructor");  
 }  
}**

**public class Client2 {  
 public static void main(String[] args) {  
*// Animal a1 = new Animal();  
// Mammal m1 = new Mammal();  
// Dog d1 = new Dog();* Dog d2 = new Dog("Golden Retriever");  
 }  
}**

**Output**

Animal zero-arg constructor

Mammal zero-arg constructor

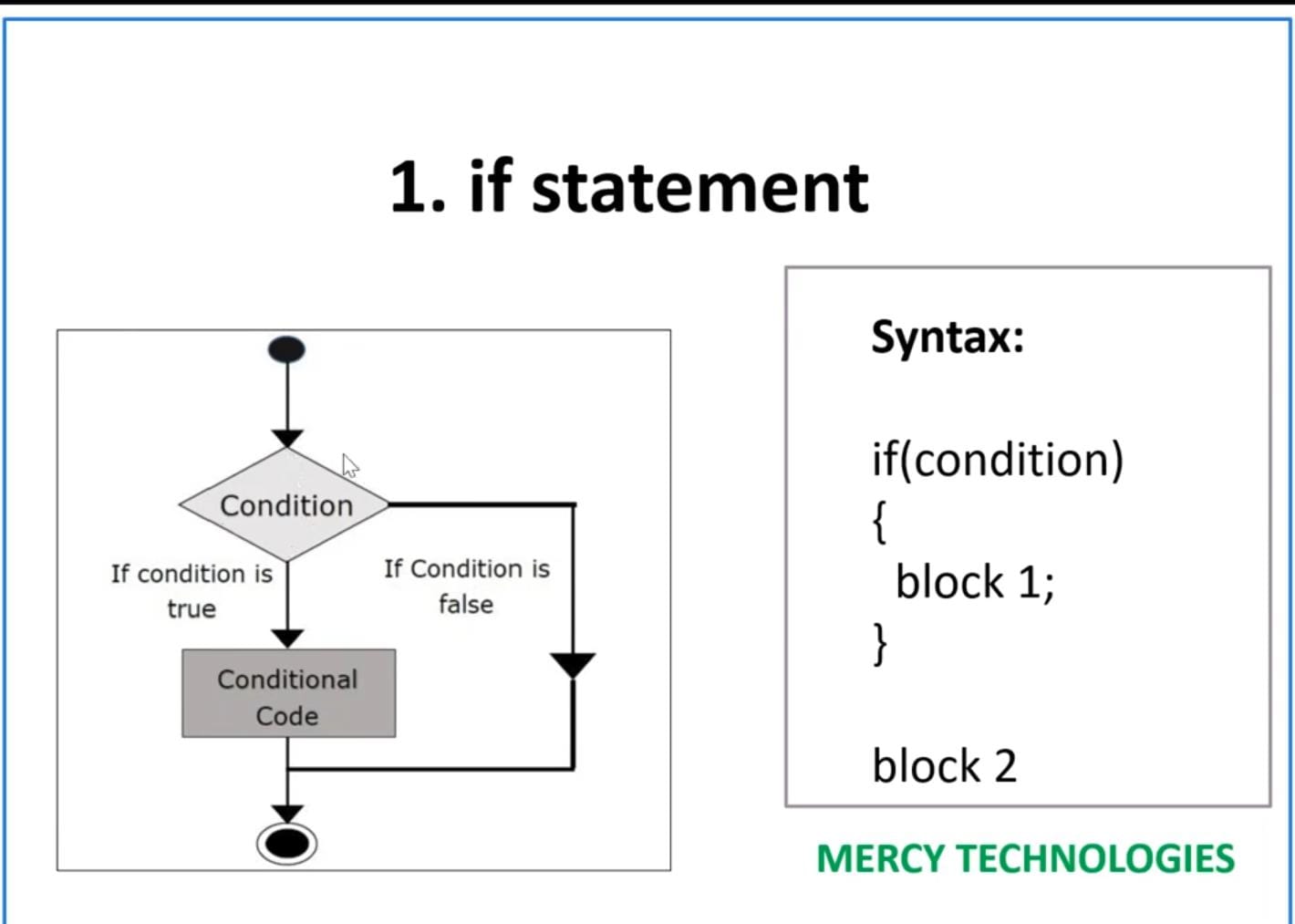
Dog zero-arg constructor

Dog parameterized constructor constructor

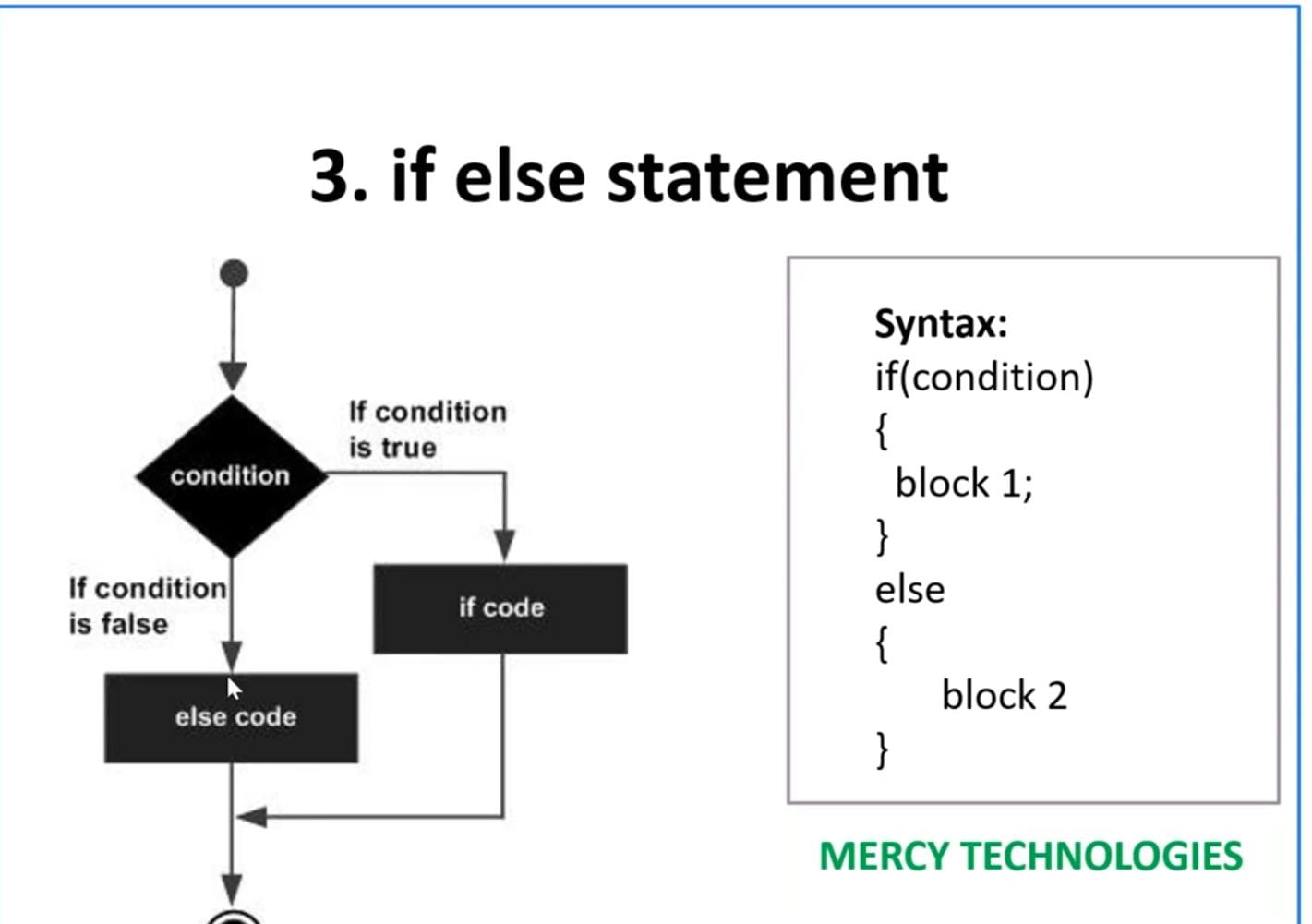
**Java Conditional Statements**



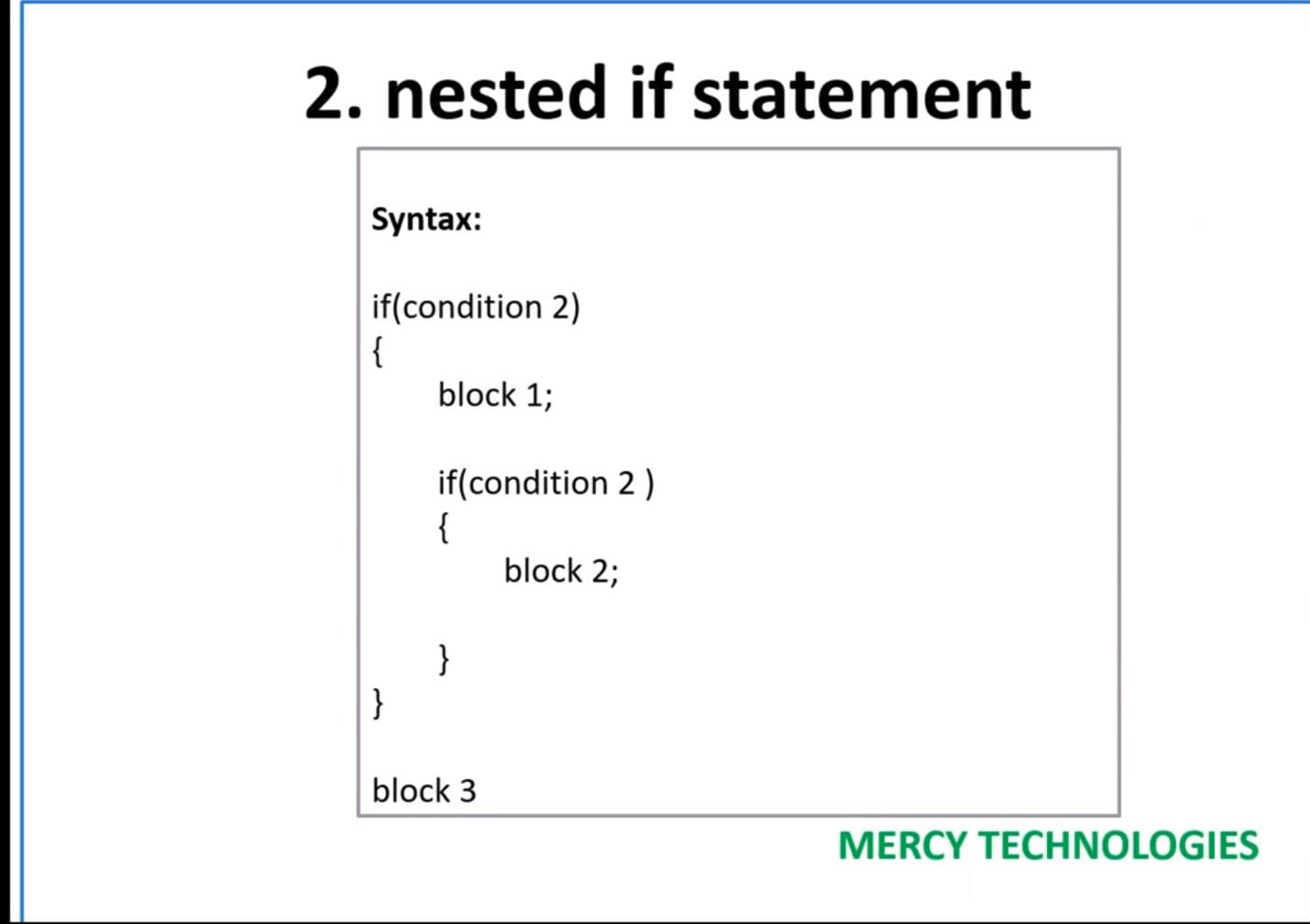
**In Java, conditional statements are used to execute different blocks of code based on certain conditions.**

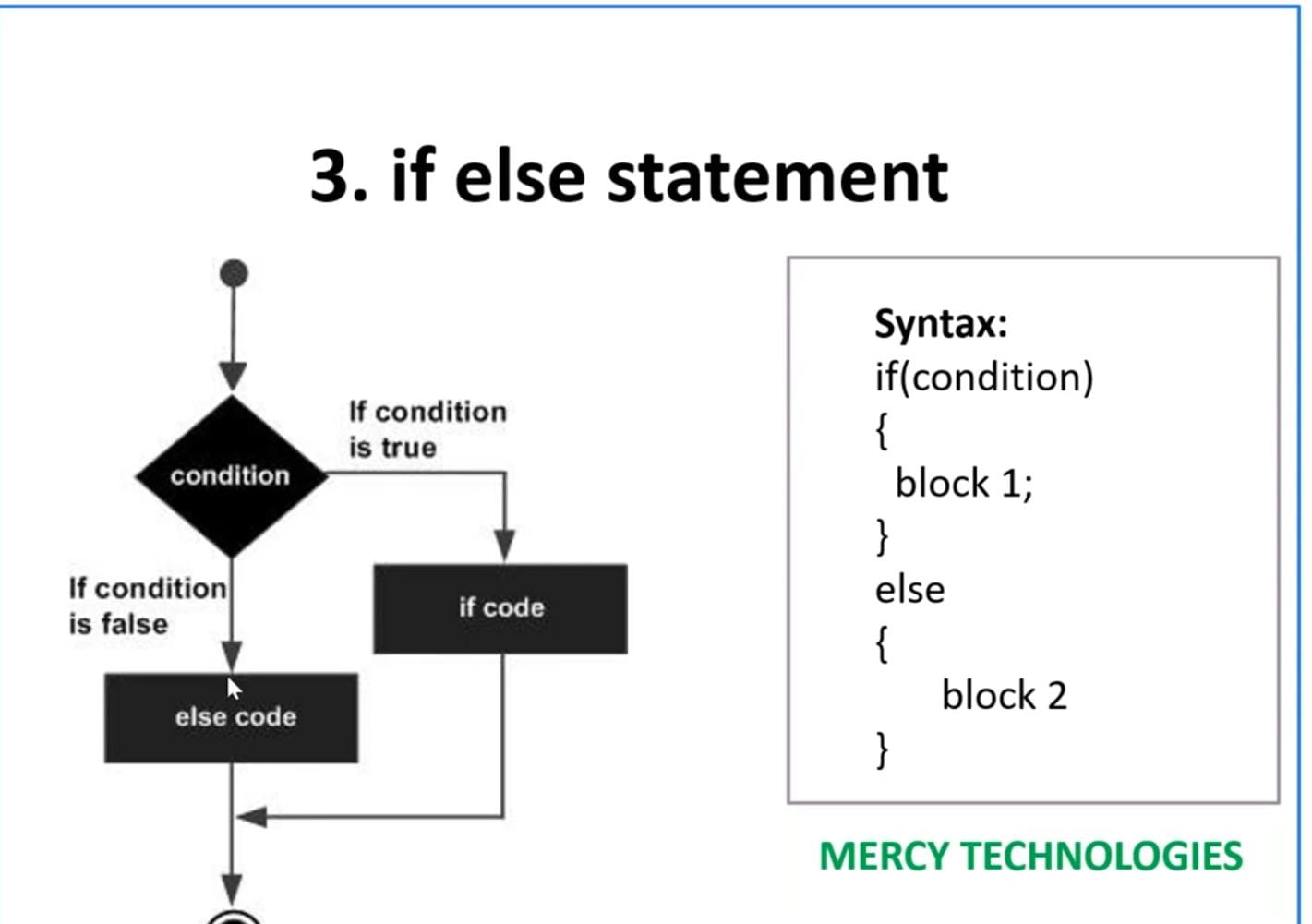


**public class ExampleIf {  
 public static void main(String[] args) {  
 int number = 10;  
  
 if (number > 0) {  
 System.*out*.println("The number is positive.");  
 }  
 }  
}**



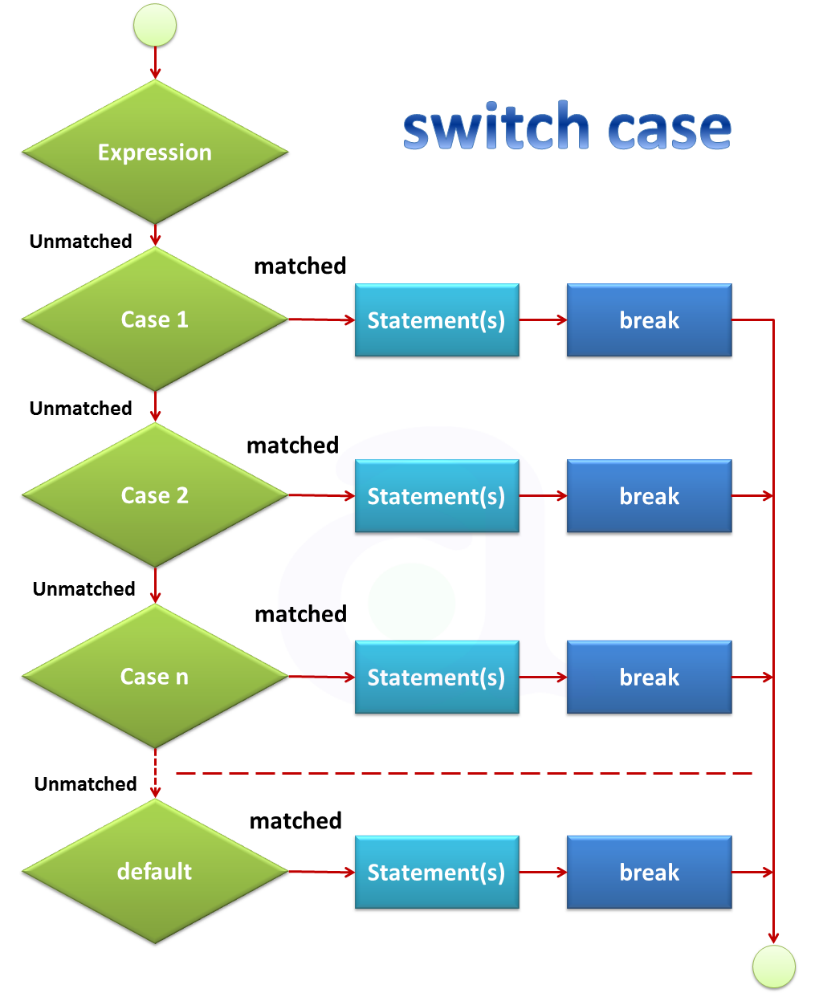
**public class ExampleIfElse {  
 public static void main(String[] args) {  
 int number = -5;  
 if (number > 0) {  
 System.*out*.println("The number is positive.");  
 } else {  
 System.*out*.println("The number is not positive.");  
 }  
 }  
}**





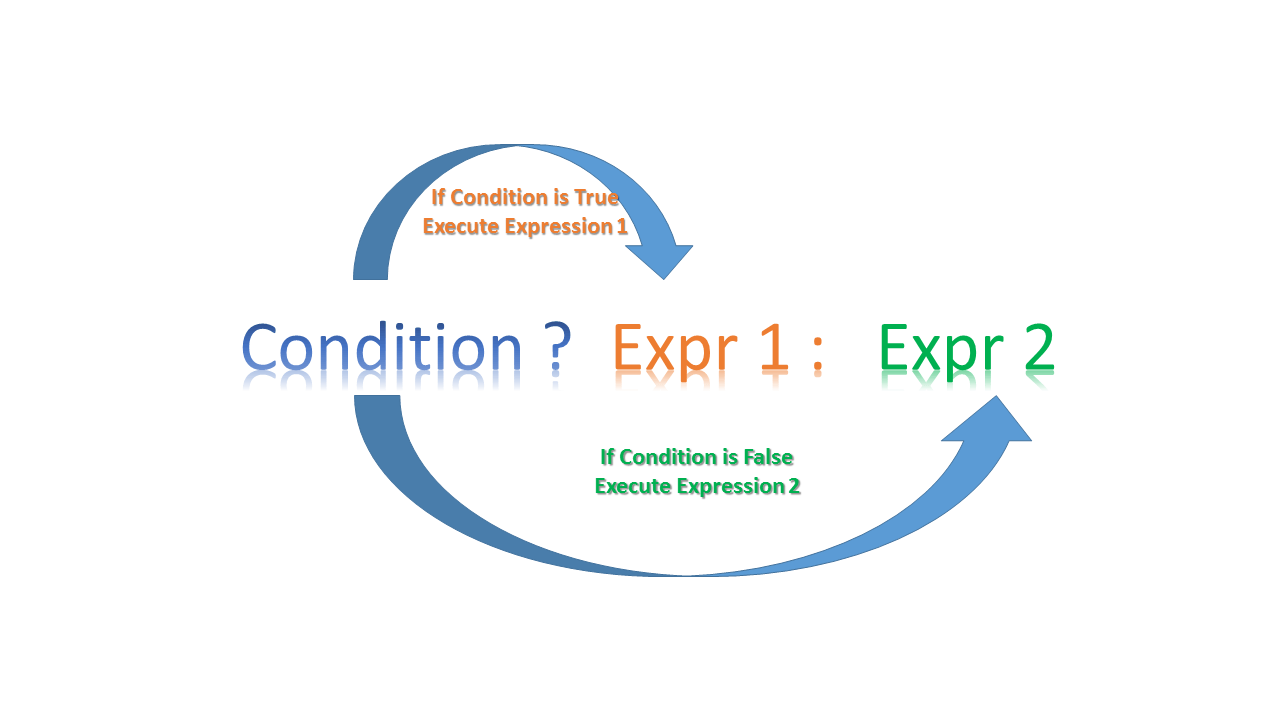
**public class ExampleIfElse {  
 public static void main(String[] args) {  
 int number = -5;  
 if (number > 0) {  
 System.*out*.println("The number is positive.");  
 } else {  
 System.*out*.println("The number is not positive.");  
 }  
 }  
}**

**Switch Statement:**



public class ExampleSwitch {  
 public static void main(String[] args) {  
 int day = 3;  
 String dayName;  
  
 switch (day) {  
 case 1:  
 dayName = "Sunday";  
 break;  
 case 2:  
 dayName = "Monday";  
 break;  
 case 3:  
 dayName = "Tuesday";  
 break;  
 case 4:  
 dayName = "Wednesday";  
 break;  
 case 5:  
 dayName = "Thursday";  
 break;  
 case 6:  
 dayName = "Friday";  
 break;  
 case 7:  
 dayName = "Saturday";  
 break;  
 default:  
 dayName = "Invalid day";  
 break;  
 }  
  
 System.*out*.println("The day is: " + dayName);  
 }  
}

Ternary Operator:



**public class ExampleTernaryOperator {  
 public static void main(String[] args) {  
 int number = 5;  
 String result = (number > 0) ? "Positive" : "Negative";  
 System.*out*.println("The number is: " + result);  
 }  
}**

**1. for Loop**

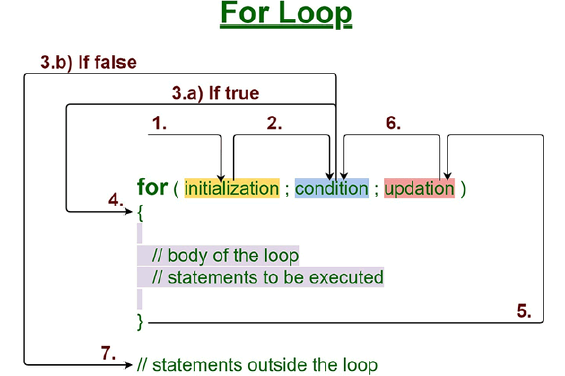
The for loop is perfect when you know in advance how many times you want to repeat a block of code.

**Structure:**

**for (initialization; condition; updation) {**

**// Code to be executed repeatedly**

**}**

****

**Example: Print numbers from 1 to 5**

**public class ForLoopExample {  
 public static void main(String[] args) {  
 for (int i = 1; i <= 5; i++) {  
 System.*out*.println("Number: " + i);  
 }  
 }  
}**

**Explanation:**

* **Initialization**: int i = 1 sets the starting point of the loop with i starting at 1.
* **Condition**: i <= 5 determines whether the loop should continue running.
* **Update**: i++ increments the value of i by 1 after each iteration.

**2. while Loop**

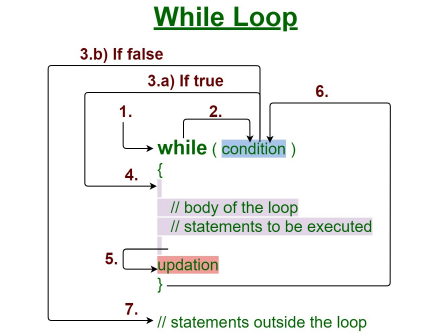
The while loop is useful when you don't know in advance how many times you want to loop, but you have a condition to keep checking.

**Structure:**

**while (condition) {**

**// Code to be executed repeatedly**

**}**

****

**Example: Print numbers from 1 to 5**

**public class WhileLoopExample {  
 public static void main(String[] args) {  
 int i = 1;  
 while (i <= 5) {  
 System.*out*.println("Number: " + i);  
 i++;  
 }  
 }  
}**

**Explanation:**

* **Condition**: i <= 5 is checked before each loop iteration; the loop continues as long as this is true.
* **Update inside the loop**: i++ increments i after each print, eventually ending the loop when i exceeds 5.

**3. do-while Loop**

The do-while loop is similar to the while loop but ensures that the loop's code runs at least once, even if the condition is false.

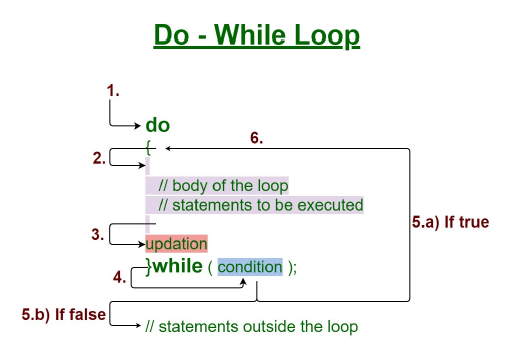
**Structure:**

**do {**

**// Code to be executed**

**} while (condition);**

**Example: Print numbers from 1 to 5**

****

**public class DoWhileLoopExample {  
 public static void main(String[] args) {  
 int i = 1;  
 do {  
 System.*out*.println("Number: " + i);  
 i++;  
 } while (i <= 5);  
 }  
}**

**Explanation:**

* **Execute First**: The block of code runs once before checking the condition.
* **Condition**: i <= 5 is evaluated after each loop iteration. If true, the loop continues; if false, the loop ends.

**4. Enhanced for Loop (for-each loop)**

The enhanced for loop simplifies iterating through elements in an array or collection, without needing to manage an index variable.

**Structure:**

**for (type var : array) {**

**// Code to be executed for each element**

**}**

**Example: Print elements of an array**

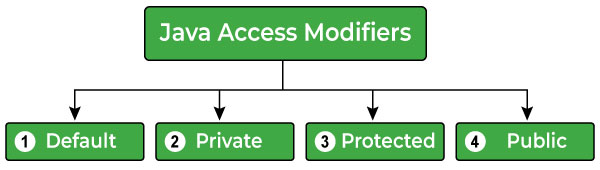
**public class ForEachLoopExample {  
 public static void main(String[] args) {  
 int[] numbers = {1, 2, 3, 4, 5};  
 for (int num : numbers) {  
 System.*out*.println("Number: " + num);  
 }  
 }  
}**

**Explanation:**

* **Element Access**: The loop automatically iterates over each element in the numbers array.
* **No Index Needed**: The current element is stored in num for each iteration, eliminating the need for an index variable.

**public class ForEachExample2 {  
 public static void main(String[] args) {  
 List<String> names = List.*of*("Mercy","Angel","Java");  
 for(String name : names){  
 System.*out*.println(name);  
 }  
 }  
}**

**Access Modifiers in Java**



1. **private**: Accessible **only within the same class**; cannot be accessed from outside.
2. **default (no modifier)**: Accessible **within the same package**, but not from outside the package.
3. **protected**: Accessible **within the same package** and by **subclasses**, even if they are in different packages. (default + kids)
4. **public**: Accessible from **anywhere**, in any class or package.

**private:** Its like your bedroom. Only **you** (the class) can access it. Even your family (other classes) or guests (subclasses) can’t enter without permission.

**default:** This is like the living room. Only people inside the house (same package) can access it. Neighbors or people from outside (different packages) can’t enter.

**protected:** This is like common amenities – like a swimming pool, club house, kids play area : **Family members** (same package) and **guests** (subclasses) with special permission can access it, even if they come from outside (different packages). Strangers can't enter.

**public:** This is like approach road to your apartment – anyone can access. Like, delivery executives, cab drivers etc. No restriction at all.

**Why do we need different access modifiers?**

* **Encapsulation**: Hide details, expose only what’s necessary.
* **Security**: Protect data from unwanted changes.
* **Modularity**: Organize code by separating internal and external use.
* **Inheritance Control**: Allow subclasses access, block unrelated classes.

**package com.mercy.access.student;  
  
public class Student {  
 *// declaring private instance variable* private int rollNo;  
 public static void main(String[] args) {  
 System.*out*.println("This is from com.mercy.access.student.Student");  
 Student student = new Student();  
 *// accessing private instance variable* System.*out*.println(student.rollNo);  
 }  
}**

**package com.mercy.access.student;  
  
public class Student2 {  
 *// declaring private instance method* private void findTotal() {  
 System.*out*.println("This is from findTotal() of Student2");  
 }  
  
 public static void main(String[] args) {  
 System.*out*.println("This is from com.mercy.access.student.Student");  
 Student2 student2 = new Student2();  
 *// accessing private instance method* student2.findTotal();  
 }  
}**

**package com.mercy.access.student;  
  
public class Student3 {  
 *// declaring private instance variable* private int rollNo;  
  
 *// declaring private instance method* private void findTotal() {  
 System.*out*.println("This is from findTotal() of Student3");  
 }  
}  
  
class Client{  
 public static void main(String[] args) {  
 Student3 student3 = new Student3();  
 student3.rollNo;*// private access* student3.findTotal(); *// private access* }  
}**

**package com.mercy.access.student;  
  
public class Student4 {  
 private String name;  
 private int rollNo;  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public int getRollNo() {  
 return rollNo;  
 }  
  
 public void setRollNo(int rollNo) {  
 this.rollNo = rollNo;  
 }  
  
 @Override  
 public String toString() {  
 return "Student4{" +  
 "name='" + name + '\'' +  
 ", rollNo=" + rollNo +  
 '}';  
 }  
}  
  
class Client4{  
 public static void main(String[] args) {  
 Student4 student4 = new Student4();  
 System.*out*.println(student4);  
 student4.setName("Mercy");  
 student4.setRollNo(100);  
 System.*out*.println(student4);  
 }  
}**

**More about private access modifier:**

* private instance variables will not be inherited to subclass.
* private instance methods will not be inherited to subclass.
* we can access private constructor within same class.
* we cannot access private constructor from child class or different class.
* We can restrict object creation from outside class if we make the constructor private.
* if one constructor is private but others are not, we can use the available construtors to create object.
* We cannot declare static block as private.
* Wecannot declare instance block as private.

private instance variables will not be inherited to subclass

**package com.mercy.access;  
  
public class Student5 {  
 boolean status;  
 private int rollNo;  
 private String studentName;  
}  
  
class Client5 extends Student5 {  
 public static void main(String[] args) {  
 Student5 student5 = new Student5();  
 System.*out*.println(student5.status);  
 *// student5.rollNo has private access***

**System.*out*.println(student5.rollNo);**

***// student5.studentName has private access* System.*out*.println(student5.studentName); }  
}**

private instance methods will not be inherited to subclass

**package com.mercy.access;  
  
public class Student6 {  
 boolean status;  
 private int rollNo;  
 private String studentName;  
  
 void findAverage(){  
 System.*out*.println("from Student6.findAverage()");  
 }  
 private void findTotal(){  
 System.*out*.println("from Student6.findTotal()");  
 }  
}  
  
class Client6 extends Student6 {  
 public static void main(String[] args) {  
 Student6 student6 = new Student6();  
 System.*out*.println(student6.status);  
*// student6.rollNo has private access  
// System.out.println(student6.rollNo);  
// student6.studentName has private access  
// System.out.println(student6.studentName);* student6.findAverage();  
*// student6.findTotal() has private access  
// student6.findTotal();* }  
}**

we can access private constructor within same class.

**public class Student7 {**

*// private constructor*

**private Student7() {  
 }  
  
 public static void main(String[] args) {  
 Student7 student7 = new Student7();  
 }  
}**

we cannot access private constructor from child class or different class.

**public class Student8 {  
 *// private constructor* private Student8() {  
 }  
}**  
  
**class Client8 extends Student8 {  
 public static void main(String[] args) {  
*// Student8() has private access;  
// Student8 student8 = new Student8();* }  
}**

if one constructor is private but others are not, we can use the available construtors to create object.

**public class Student9 {  
 private Student9() {  
 }  
  
 Student9(int rollNo) {  
  
 }  
}  
  
class Client9 {  
 public static void main(String[] args) {  
 Student9 student9 = new Student9(100);  
 }  
}**

**Protected Access Modifier:**

I- Usage Level

II- Inheritance Level

1- if we do not specify any keyword, then the access level is defult access level.

2- default variables can be accessed outside the class,

but within the same package.

3- default methods can be accessed outside the class,

but within the same package.

4- All the default variables will be inherited to the sub class

so, we can access them using the reference variable to sub class

5- All the default methods will be inherited to the sub class

so, we can access them using the reference variable to sub class

6- default instance members

will be inherited to subclasses with same package.

7- default instance members

will NOT be inherited to subclasses outside the package.

**package com.mercy.access.employee;  
  
*// class is having default access level*public class Employee1 {  
 *// instance variables are having default level* int empId;  
 String empName;  
}**

**package com.mercy.access.employee;**

**public class Client1 {  
 public static void main(String[] args) {  
 Employee1 employee1 = new Employee1();  
 System.*out*.println(employee1.empId);  
 System.*out*.println(employee1.empName);  
 }  
}**

**package com.mercy.access.employee;**

***// class is having default access level*public class Employee2 {  
 *// instance method is having default level* void printEmployeeDetails() {  
 System.*out*.println("from printEmployeeDetails()");  
 }  
}**

**package com.mercy.access.employee;**

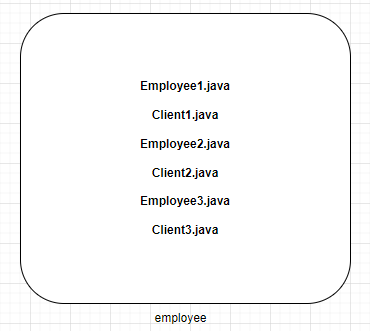
**public class Client2 {  
 public static void main(String[] args) {  
 Employee2 employee2 = new Employee2();  
 employee2.printEmployeeDetails();  
 }  
}**

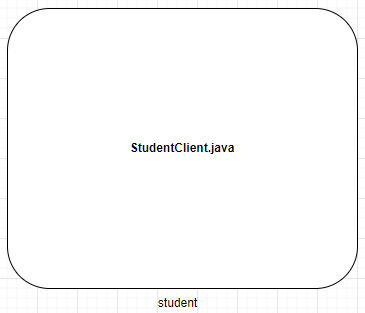
**package com.mercy.access.employee;**

***// class is having default access level*public class Employee3 {  
 *// instance variables are having default level* int empId;  
 String empName;  
}**

**package com.mercy.access.employee;**

**public class Client3 extends Employee3{  
 public static void main(String[] args) {  
 Client3 client3 = new Client3();  
 System.*out*.println(client3.empId);  
 System.*out*.println(client3.empName);  
 }  
}**



****

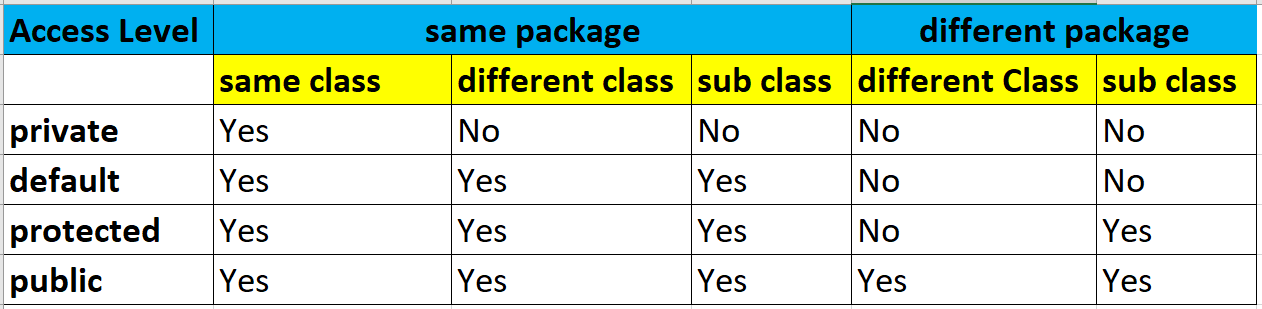
**package com.mercy.access.student;  
  
import com.mercy.access.employee.Employee1;  
  
class StudentClient {  
 public static void main(String[] args) {**

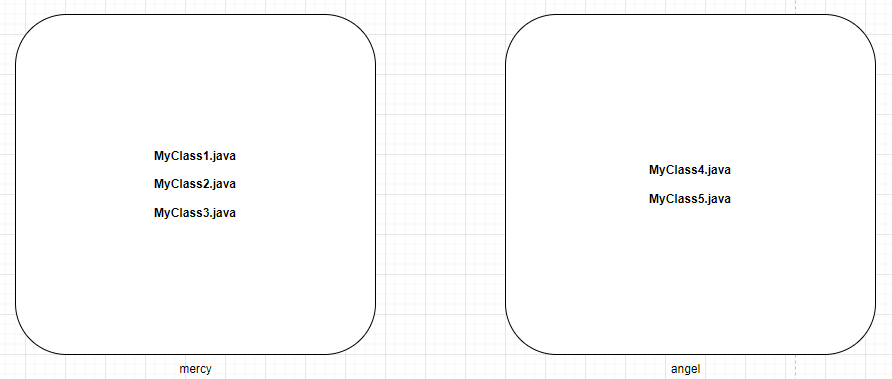
***// this is possible only if Employee1 is public*  
 Employee1 employee1 = new Employee1();  
 }  
}**

**package com.mercy.access.student;  
  
import com.mercy.access.employee.Employee1;  
  
class StudentClient {  
 public static void main(String[] args) {  
 Employee1 employee1 = new Employee1();  
 }  
}**

**package com.mercy.access.student;  
  
import com.mercy.access.employee.Employee1;  
  
public class StudentClient1 extends Employee1 {  
 public static void main(String[] args) {  
 Employee1 employee1 = new Employee1();**

***// this is NOT possible even after using extends keyword*  
 employee1.empId;  
 }  
}**





**package com.mercy.access.mercy;  
  
public class MyClass1 {  
 private int p; *// private instance variable* int q; *// default instance variable* protected int r; *// protected instance variable* public int s; *// public instance variable* public static void main(String[] args) {  
 MyClass1 m1 = new MyClass1();  
 System.*out*.println(m1.p);  
 System.*out*.println(m1.q);  
 System.*out*.println(m1.r);  
 System.*out*.println(m1.s);  
 }  
}**

**package com.mercy.access.mercy;  
  
public class MyClass2 {  
 public static void main(String[] args) {  
 MyClass1 m1 = new MyClass1();  
*// System.out.println(m1.p); // 'p' has private access in MyClass1* System.*out*.println(m1.q);  
 System.*out*.println(m1.r);  
 System.*out*.println(m1.s);  
 }  
}**

**package com.mercy.access.mercy;  
  
public class MyClass3 extends MyClass1 {  
 public static void main(String[] args) {  
 MyClass3 m1 = new MyClass3();  
*// System.out.println(m1.p); // 'p' has private access in MyClass1* System.*out*.println(m1.q);  
 System.*out*.println(m1.r);  
 System.*out*.println(m1.s);  
 }  
}**

**package com.mercy.access.angel;  
  
import com.mercy.access.mercy.MyClass1;  
  
public class MyClass4 {  
 public static void main(String[] args) {  
 MyClass1 m1 = new MyClass1();  
*// System.out.println(m1.p); // 'p' has private access in MyClass1  
// System.out.println(m1.q); // 'q' has default access in MyClass1  
// System.out.println(m1.r); // 'r' has protected access in MyClass1* System.*out*.println(m1.s);  
 }  
}**

**package com.mercy.access.angel;  
  
import com.mercy.access.mercy.MyClass1;  
  
public class MyClass5 extends MyClass1 {  
 public static void main(String[] args) {  
 MyClass5 m1 = new MyClass5();  
*// System.out.println(m1.p); // 'p' has private access in MyClass1  
// System.out.println(m1.q); // 'q' has default access in MyClass1* System.*out*.println(m1.r);  
 System.*out*.println(m1.s);  
 }  
}**

**Type Casting**

1. Widening (implicit)
2. Narrowing

**byte < short < int < long < float < double**

**1Kg < 5Kg < 10Kg < 50Kg < 150.5 Kg < 500.25Kg**

**package com.mercy.type.casting;  
  
public class Client1 {  
 public static void main(String[] args) {  
 byte b = 100; *// 100 placed in 1Kg bag* short s = b; *// 100 placed in 5Kg bag (implicit widening)* System.*out*.println(b);  
 System.*out*.println(s);  
 }  
}**

**package com.mercy.type.casting;  
  
public class Client2 {  
 public static void main(String[] args) {  
 short s = 1000;  
 int i = s; *// (implicit widening)* System.*out*.println(s);  
 System.*out*.println(i);  
 }  
}**

**package com.mercy.type.casting;  
  
public class Client3 {  
 public static void main(String[] args) {  
 byte b = 40;  
 long l = b; *// implicit widening* System.*out*.println(b);  
 System.*out*.println(l);  
 }  
}**

**package com.mercy.type.casting;  
  
public class Client4 {  
 public static void main(String[] args) {  
 float f = 150.25f;  
 double d = f; *// implicit widening* System.*out*.println(f);  
 System.*out*.println(d);  
 }  
}**

**package com.mercy.type.casting;  
  
public class Client5 {  
 public static void main(String[] args) {  
 byte b = 100;  
 short s = b; *// implicit widening -> byte to short* int i = s; *// implicit widening -> short to int* long l = i; *// implicit widening -> int to long* System.*out*.println(b);  
 System.*out*.println(s);  
 System.*out*.println(i);  
 System.*out*.println(l);  
 }  
}**

**package com.mercy.type.casting;  
  
public class Client6 {  
 private static void convert(int number) {  
 System.*out*.println("number : " + number);  
 }  
  
 public static void main(String[] args) {  
 byte b = 100;  
 *convert*(b); *// implicit widening -> byte to int* short s = 200;  
 *convert*(s); *// implicit widening -> short to int* int i = 300;  
 *convert*(i); *// no type casting happens* int i1 = s; *// implicit widening -> short to int  
 convert*(i1); *// no type casting happens* }  
}**

**package com.mercy.type.casting;  
  
public class Client7 {  
 private static float process(long number){  
 return number; *// implicit widening -> long to float* }  
 public static void main(String[] args) {  
 byte b = 100;  
 float f = *process*(b); *// no type casting happens* short s = 200;  
 double d = *process*(s); *// implicit widening -> float to double* }  
}**

**Explicit Narrowing:**

**byte > short > int > long > float > double**

**1Kg > 5Kg > 10Kg > 50Kg > 150.5 Kg > 500.25Kg**

What is explicit Narrowing?

* Converting from a broader data type to a narrower data type is called narrowing conversion

What is the disadvantage?

* Possible data loss.



What is the advantage?

* Better usage of space.

**package com.mercy.type.casting.narrowing;  
  
public class Client1 {  
 public static void main(String[] args) {  
 int i = 100000;  
 long l = i; *// Auto Widening* System.*out*.println(i);  
 System.*out*.println(l);  
  
 short s = i; *// Explicit Narrowing* }  
}**

**package com.mercy.type.casting.narrowing;  
  
public class Client2 {  
 public static void main(String[] args) {  
 int i = 100000;  
 long l = i; *// Auto Widening* System.*out*.println(i); *// 100000* System.*out*.println(l); *// 100000* short s = (short) i; *// Type Casting - Explicit Narrowing* System.*out*.println(i); *// 100000* System.*out*.println(s); *// -31072* byte b = (byte) i; *// Type Casting - Explicit Narrowing* System.*out*.println(i); *// 100000* System.*out*.println(b); *// -96* }  
}**

**package com.mercy.type.casting.narrowing;  
public class Client3 {  
 public static void main(String[] args) {  
 int i = 100;  
 long l = i; *// Auto Widening* System.*out*.println(i); *// 100* System.*out*.println(l); *// 100* short s = (short) i; *// Type Casting - Explicit Narrowing* System.*out*.println(i); *// 100* System.*out*.println(s); *// 100* byte b = (byte) i; *// Type Casting - Explicit Narrowing* System.*out*.println(i); *// 100* System.*out*.println(b); *// 100* }  
}**

**package com.mercy.type.casting.narrowing;  
  
public class Client4 {  
 public static void main(String[] args) {  
 long l = 100000;  
 int i = (int) l; *// Type Casting - Explicit Narrowing - long to int* short s = (short) i; *// Type Casting - Explicit Narrowing - int to short* byte b = (byte) s; *// Type Casting - Explicit Narrowing - short to byte* System.*out*.println(l); *// 100000* System.*out*.println(i); *// 100000* System.*out*.println(s); *// -31072* System.*out*.println(b); *// -96* }  
}**

**package com.mercy.type.casting.narrowing;  
  
public class Client5 {  
 private static void convert(byte number) {  
 System.*out*.println("number : " + number); *// number : 16* }  
  
 public static void main(String[] args) {  
 int i = 10000;  
 *convert*((byte) i); *// Type Casting - Explicit Narrowing - int to byte* }  
}**

**package com.mercy.type.casting.narrowing;  
  
public class Client6 {  
 private static void convert(byte number) {  
 System.*out*.println("number : " + number); *// number : 100* }  
  
 public static void main(String[] args) {  
 int i = 100;  
 *convert*((byte) i); *// Type Casting - Explicit Narrowing - int to byte* }  
}**

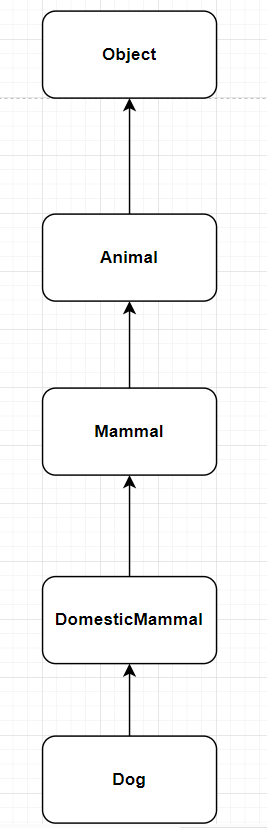
**package com.mercy.type.casting.narrowing;  
  
public class Client7 {  
 *// this is auto widening* private static double process(float number) {  
 System.*out*.println("number : " + number);  
 return number;  
 }  
  
 *// this is explicit narrowing* private static float process1(double number) {  
 System.*out*.println("number : " + number);  
 return (float) number;  
 }  
  
 public static void main(String[] args) {  
 *process*(100.55f);  
 *process1*(550.56d);  
 }  
}**

**Auto Upcasting**

* Auto Upcasting: (similar to Auto Widening).

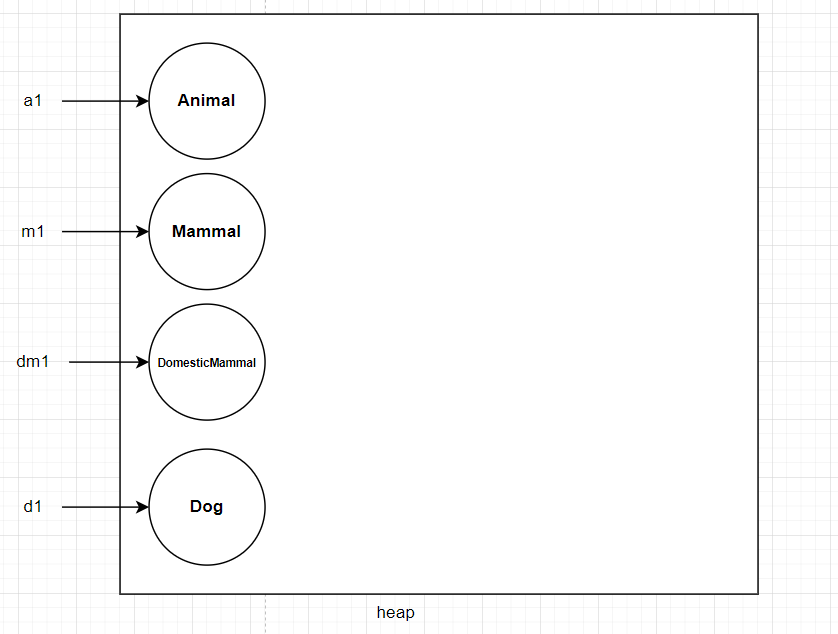
Storing child object in super class / super interface

* Upcasting is allowed by default.



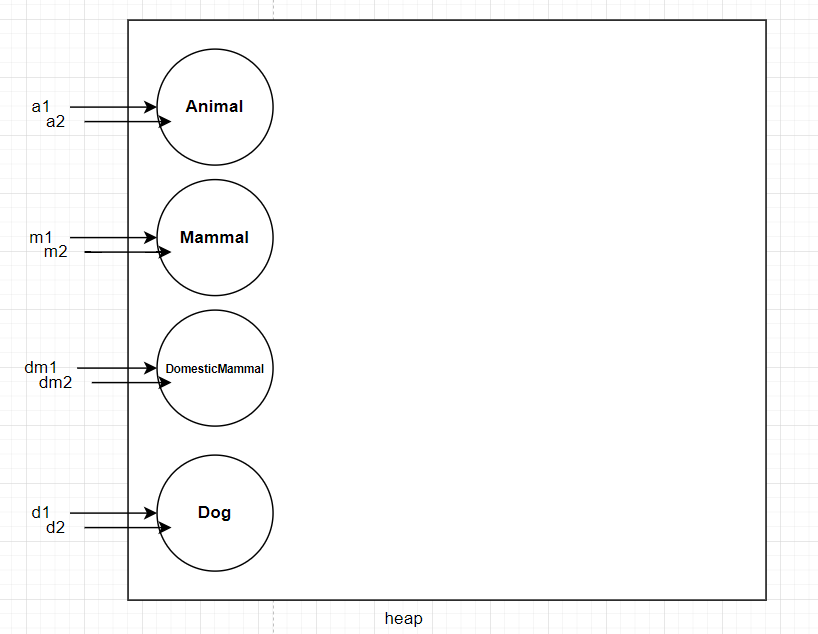
**package com.mercy.type.casting.upcasting;  
  
public class Animal {  
}  
  
class Mammal extends Animal {  
  
}  
  
class DomesticMammal extends Mammal {  
  
}  
  
class Dog extends DomesticMammal {  
  
}**

**package com.mercy.type.casting.upcasting;  
  
public class Client1 {  
 public static void main(String[] args) {  
 Animal a1 = new Animal();  
 Mammal m1 = new Mammal();  
 DomesticMammal dm1 = new DomesticMammal();  
 Dog d1 = new Dog();  
 }  
}**



**package com.mercy.type.casting.upcasting;  
  
public class Client2 {  
 public static void main(String[] args) {  
 Animal a1 = null;  
 Mammal m1 = null;  
 DomesticMammal dm1 = null;  
 Dog d1 = null;  
  
 a1 = new Animal();  
 m1 = new Mammal();  
 dm1 = new DomesticMammal();  
 d1 = new Dog();  
 }  
}**

**package com.mercy.type.casting.upcasting;  
  
public class Client3 {  
 public static void main(String[] args) {  
 Animal a1, a2 = null;  
 Mammal m1, m2 = null;  
 DomesticMammal dm1, dm2 = null;  
 Dog d1, d2 = null;  
  
 a1 = new Animal();  
 a2 = a1;  
  
 m1 = new Mammal();  
 m2 = m1;  
  
 dm1 = new DomesticMammal();  
 dm2 = dm1;  
  
 d1 = new Dog();  
 dm2 = d1;  
 }  
}**



**package com.mercy.type.casting.upcasting;  
  
public class Client4 {  
 public static void main(String[] args) {  
 Object o1 = new Object();  
 Animal a1 = new Animal();  
 Mammal m1 = new Mammal();  
 DomesticMammal dm1 = new DomesticMammal();  
 Dog d1 = new Dog();  
  
 *// calling using reference variable  
 process*(o1);  
 *process1*(a1);  
 *process2*(m1);  
 *process3*(dm1);  
 *process4*(d1);  
  
 *// calling using the object  
 process2*(new Mammal());  
 *process4*(new Dog());  
 }  
  
 public static void process(Object obj1) {  
 System.*out*.println("process(Object obj1)");  
 }  
  
 public static void process1(Animal obj1) {  
 System.*out*.println("process1(Animal obj1)");  
 }  
  
 public static void process2(Mammal obj1) {  
 System.*out*.println("process2(Mammal obj1)");  
 }  
  
 public static void process3(DomesticMammal obj1) {  
 System.*out*.println("process3(DomesticMammal obj1)");  
 }  
  
 public static void process4(Dog obj1) {  
 System.*out*.println("process4(Dog obj1)");  
 }  
}**

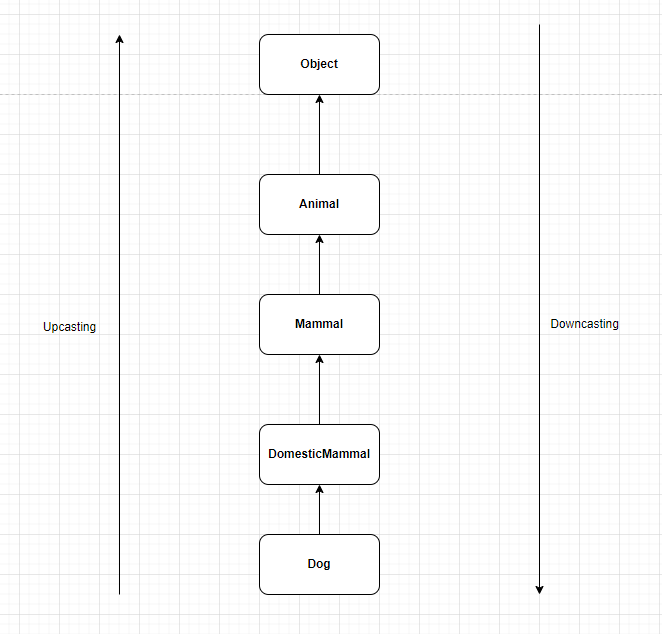
**package com.mercy.type.casting.upcasting;  
public class Client5 {  
 public static void main(String[] args) {  
 Object o1 = null;  
 Animal a1 = null;  
 Mammal m1 = null;  
 DomesticMammal dm1 = null;  
 Dog d1 = null;  
  
 o1 = *process*();  
 a1 = *process1*();  
 m1 = *process2*();  
 dm1 = *process3*();  
 d1 = *process4*();  
 }  
  
 public static Object process() {  
 System.*out*.println("process(Object obj1)");  
 return new Object();  
 }  
  
 public static Animal process1() {  
 System.*out*.println("process1(Animal obj1)");  
 return new Animal();  
 }  
  
 public static Mammal process2() {  
 System.*out*.println("process2(Mammal obj1)");  
 return new Mammal();  
 }  
  
 public static DomesticMammal process3() {  
 System.*out*.println("process3(DomesticMammal obj1)");  
 return new DomesticMammal();  
 }  
  
 public static Dog process4() {  
 System.*out*.println("process4(Dog obj1)");  
 return new Dog();  
 }  
}**

**Program to explain Auto Upcasting**

**package com.mercy.type.casting.upcasting;  
  
public class Client6 {  
 public static void main(String[] args) {  
 Object o1, o2 = null;  
 Animal a1, a2 = null;  
 Mammal m1, m2 = null;  
 DomesticMammal dm1 = null;  
 Dog d1 = null;  
  
 o1 = *process*();  
 a1 = *process1*();  
 m1 = *process2*();  
 dm1 = *process3*();  
 d1 = *process4*();  
  
 o2 = *process4*();  
 a2 = *process3*();  
 m2 = *process4*();  
 }  
  
 public static Object process() {  
 System.*out*.println("process(Object obj1)");  
 return new Object();  
 }  
  
 public static Animal process1() {  
 System.*out*.println("process1(Animal obj1)");  
 return new Animal();  
 }  
  
 public static Mammal process2() {  
 System.*out*.println("process2(Mammal obj1)");  
 return new Mammal();  
 }**

**public static DomesticMammal process3() {  
 System.*out*.println("process3(DomesticMammal obj1)");  
 return new DomesticMammal();  
 }  
  
 public static Dog process4() {  
 System.*out*.println("process4(Dog obj1)");  
 return new Dog();  
 }  
}**

**Explicit Downcasting:**



**1- What is explicit downcasting?**

* Explicit Downcasting: (similar to Explicit Narrowing).

Storing parent object in child class reference

* Explicit Downcasting - this is not allowed by default.

If not downcaseted perperly, we get run time exception **ClassCastException**

**package com.mercy.type.casting.downcasting;  
  
public class Class6 {  
 public static void main(String[] args) {  
 Dog d1 = new Dog();  
 *process2*(d1);  
  
 Animal a1 = new Dog();  
 *process2*((Dog) a1);  
  
 Animal a2 = new Animal();  
 *process2*((Dog) a2);  
 }  
  
 private static void process2(Dog obj1) {  
 System.*out*.println("process2(Dog obj1)");  
 }  
}**

**package com.mercy.type.casting.downcasting;  
  
public class Client1 {  
 public static void main(String[] args) {  
 *// Auto Upcasting* Animal a1 = new Animal();  
 Animal a2 = new Mammal();  
 Animal a3 = new DomesticMammal();  
 Animal a4 = new Dog();  
  
 */\*  
 \* Animal a1 = new Animal();  
 \* Animal a2 = (Animal) new Mammal();  
 \* Animal a3 = (Animal) new DomesticMammal();  
 \* Animal a4 = (Animal) new Dog();  
 \* \*/* }  
}**

**package com.mercy.type.casting.downcasting;  
  
public class Client2 {  
 public static void main(String[] args) {  
 Animal a1 = new Mammal(); *// Auto Upcasting* Mammal m1 = (Mammal) new Animal(); *// Explicit down casting* }  
}**

**package com.mercy.type.casting.downcasting;  
  
public class Client3 {  
 public static void main(String[] args) {  
 *// Auto Upcasting* Animal a2 = new Mammal();  
 Animal a3 = new DomesticMammal();  
 Animal a4 = new Dog();  
  
 *// Explicit Down casting* Dog d1 =(Dog) new Animal();  
 Dog d2 =(Dog) new Mammal();  
 Dog d3 =(Dog) new DomesticMammal();  
 }  
}**

**package com.mercy.type.casting.downcasting;  
  
public class Client4 {  
 public static void main(String[] args) {  
 Animal a1 = new Animal();  
 Dog d1 = new Dog();  
  
 a1 = d1; *// Auto Upcasting* d1 = (Dog) a1; *//* }  
}**

**package com.mercy.type.casting.downcasting;**

**public class Class6 {  
 public static void main(String[] args) {  
 Dog d1 = new Dog();  
 *process2*(d1);  
  
 Animal a1 = new Dog();  
 *process2*((Dog) a1);  
  
 Animal a2 = new Animal();  
 *process2*((Dog) a2);  
 }  
  
 private static void process2(Dog obj1) {  
 System.*out*.println("process2(Dog obj1)");  
 }  
}**

**instanceOf** Operator:

* The instnceOf operator is used to test if object is an instance of the specified type (class or subclass or interface)
* it returns boolean type (trur / false)
* if we apply instanceOf operator on null, then it always returns false.

**package com.mercy.type.casting.instance.of;  
  
public class Animal {  
}  
  
class Mammal extends Animal {  
  
}  
  
class DomesticMammal extends Mammal {  
  
}  
  
class Dog extends DomesticMammal {  
  
}**

**package com.mercy.type.casting.instance.of;  
  
public class Client1 {  
 public static void main(String[] args) {  
 Animal a1 = new Animal();  
 System.*out*.println(a1 instanceof Animal); *// true* System.*out*.println(null instanceof Animal); *// false* System.*out*.println(null instanceof Object); *// false* }  
}**

**package com.mercy.type.casting.instance.of;  
public class Client2 {  
 public static void main(String[] args) {  
 Animal a1 = new Animal();  
 Animal a2 = new Mammal();  
 Animal a3 = new DomesticMammal();  
 Animal a4 = new Dog();  
  
 System.*out*.println(a1 instanceof Animal); *// true* System.*out*.println(a2 instanceof Animal); *// true* System.*out*.println(a3 instanceof Animal); *// true* System.*out*.println(a4 instanceof Animal);*// true* }  
}**

**package com.mercy.type.casting.instance.of;  
public class Client3 {  
 public static void main(String[] args) {  
 Animal a1 = new Animal();  
 Animal a2 = new Mammal();  
 Animal a3 = new DomesticMammal();  
 Animal a4 = new Dog();  
  
 System.*out*.println(a1 instanceof Animal); *// true* System.*out*.println(a1 instanceof Mammal); *// false* System.*out*.println(a1 instanceof DomesticMammal); *// false* System.*out*.println(a1 instanceof Dog); *// false* System.*out*.println("====================");  
 System.*out*.println(a2 instanceof Animal); *// true* System.*out*.println(a2 instanceof Mammal); *// true* System.*out*.println(a2 instanceof DomesticMammal); *// false* System.*out*.println(a2 instanceof Dog); *// false* System.*out*.println("====================");  
 System.*out*.println(a3 instanceof Animal); *// true* System.*out*.println(a3 instanceof Mammal); *// true* System.*out*.println(a3 instanceof DomesticMammal); *// true* System.*out*.println(a3 instanceof Dog); *// false* System.*out*.println("====================");  
 System.*out*.println(a4 instanceof Animal); *// true* System.*out*.println(a4 instanceof Mammal); *// true* System.*out*.println(a4 instanceof DomesticMammal); *// true* System.*out*.println(a4 instanceof Dog); *// true* }  
}**

**package com.mercy.type.casting.instance.of;  
  
public class Client4 {  
 public static void process(Animal obj){  
 System.*out*.println("process(Animal obj)");  
 Dog d1 =(Dog) obj; *// down casting to Doc* }  
 public static void main(String[] args) {  
 Dog d1 = new Dog();  
 *process*(d1); *// upcasting to Animal* }  
}**

**package com.mercy.type.casting.instance.of;  
  
public class Client5 {  
 public static void process(Animal obj){  
 System.*out*.println("process(Animal obj)");  
 Dog d1 = (Dog) obj; *// down casting to Doc* }  
 public static void main(String[] args) {  
 Animal a1 = new Mammal();  
 *process*(a1); *// upcasting to Animal* }  
}**

**package com.mercy.type.casting.instance.of;  
  
public class Client6 {  
 public static void process(Animal obj) {  
 System.*out*.println("process(Animal obj)");  
 if (obj instanceof Dog) {  
 System.*out*.println("it is down casted safely to Dog");  
 Dog d1 = (Dog) obj; *// down casting to Doc* }  
 if (obj instanceof Mammal) {  
 System.*out*.println("it is down casted safely to Mammal");  
 Mammal m1 = (Mammal) obj; *// down casting to Doc* }  
 }  
  
 public static void main(String[] args) {  
 Animal a1 = new Mammal();  
 *process*(a1); *// upcasting to Animal* }  
}**

**package com.mercy.type.casting.instance.of;  
  
public class Client7 {  
 public static void process(Object obj) {  
 System.*out*.println("process(Animal obj)");  
 if (obj instanceof Animal) {  
 System.*out*.println("it is down casted safely to Animal");  
 Animal a1 = (Animal) obj;  
 }  
 if (obj instanceof Mammal) {  
 System.*out*.println("it is down casted safely to Mammal");  
 Mammal m1 = (Mammal) obj;  
 }  
 if (obj instanceof DomesticMammal) {  
 System.*out*.println("it is down casted safely to DomesticMammal");  
 DomesticMammal dm1 = (DomesticMammal) obj;  
 }  
 if (obj instanceof Dog) {  
 System.*out*.println("it is down casted safely to Dog");  
 Dog d1 = (Dog) obj;  
 }  
 }  
  
 public static void main(String[] args) {  
 Animal a1 = new Animal();  
 Animal a2 = new Mammal();  
 Animal a3 = new DomesticMammal();  
 Animal a4 = new Dog();  
  
 *process*(a1);  
 System.*out*.println("=====================");  
 *process*(a2);  
 System.*out*.println("=====================");  
 *process*(a3);  
 System.*out*.println("=====================");  
 *process*(a4);  
 System.*out*.println("=====================");  
 }  
}**

**Output:**

process(Animal obj)

it is down casted safely to Animal

=====================

process(Animal obj)

it is down casted safely to Animal

it is down casted safely to Mammal

=====================

process(Animal obj)

it is down casted safely to Animal

it is down casted safely to Mammal

it is down casted safely to DomesticMammal

=====================

process(Animal obj)

it is down casted safely to Animal

it is down casted safely to Mammal

it is down casted safely to DomesticMammal

it is down casted safely to Dog

=====================

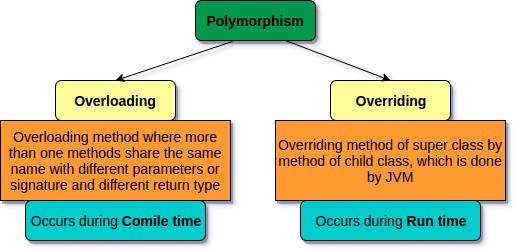
**Polymorphism**

**Definition:**  
Polymorphism is the ability of an object to take on many forms. In Java, it allows one interface to be used for a general class of actions, with the specific action determined by the exact nature of the situation.

**Types of Polymorphism:**

1. **Compile-Time Polymorphism (Method Overloading):**

* Achieved by defining multiple methods with the same name but different parameters (different number or types of arguments).
* The method to be called is determined at compile time.



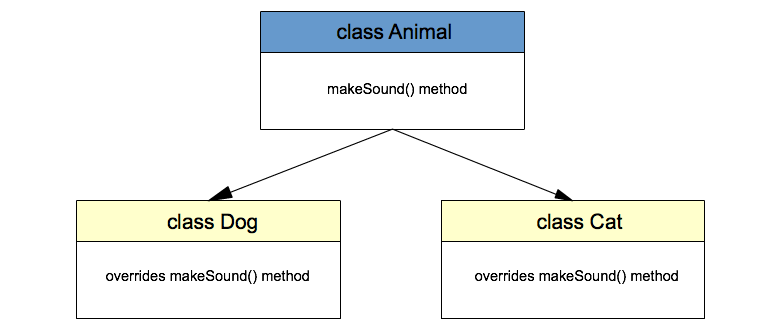
**package com.mercy.polymorphim;  
public class Addition {  
 public void add(int a, int b) {  
 System.*out*.println("add(int a, int b)");  
 System.*out*.println("sum = " + (a + b));  
 }  
 public void add(float a, float b) {  
 System.*out*.println("add(float a, float b)");  
 System.*out*.println("sum = " + (a + b));  
 }  
 public void add(String a, String b) {  
 System.*out*.println("add(String a, String b)");  
 System.*out*.println("sum = " + (a + b));  
 }  
}  
  
class AdditionClient{  
 public static void main(String[] args) {  
 Addition addition = new Addition();  
 addition.add(10,20);  
 addition.add(10.5f,20.5f);  
 addition.add("Mercy", "Technologies");  
 }  
}**

**package com.mercy.polymorphim;  
public class Addition {  
 public void add(int a, int b) {  
 System.*out*.println("add(int a, int b)");  
 System.*out*.println("sum = " + (a + b));  
 }  
 public void add(float a, float b) {  
 System.*out*.println("add(float a, float b)");  
 System.*out*.println("sum = " + (a + b));  
 }  
 public void add(String a, String b) {  
 System.*out*.println("add(String a, String b)");  
 System.*out*.println("sum = " + (a + b));  
 }  
}  
  
class AdditionClient{  
 public static void main(String[] args) {  
 Addition addition = new Addition();  
 addition.add(10,20);  
 addition.add(10.5f,20.5f);  
 addition.add("Mercy", "Technologies");  
 }  
}**

**Run-Time Polymorphism (Method Overriding):**

* Achieved when a subclass provides a specific implementation of a method already defined in its superclass.
* The method to be invoked is determined at runtime, based on the object type.





**package com.mercy.polymorphim;  
  
public class Animal {  
 public void makeSound() {  
 System.*out*.println("Some Animal Sound");  
 }  
}  
  
class Dog extends Animal {  
 public void makeSound1() {  
 System.*out*.println("Woof Woof");  
 }  
}  
  
class Cat extends Animal {  
 public void makeSound2() {  
 System.*out*.println("Meow Meow");  
 }  
}**

**package com.mercy.polymorphim;  
  
public class Client1 {  
 public static void main(String[] args) {  
 Animal a1 = new Animal();  
 a1.makeSound();  
  
 Animal a2 = new Dog();  
 a2.makeSound1();  
  
 Animal a3 = new Cat();  
 a3.makeSound2();  
 }  
}**

**package com.mercy.polymorphim;  
  
public class Animal1 {  
 public void makeSound() {  
 System.*out*.println("Some Animal Sound");  
 }  
}  
class Dog1 extends Animal1 {  
 @Override  
 public void makeSound() {  
 System.*out*.println("Woof Woof");  
 }  
}  
  
class Cat1 extends Animal1 {  
 @Override  
 public void makeSound() {  
 System.*out*.println("Meow Meow");  
 }  
}**

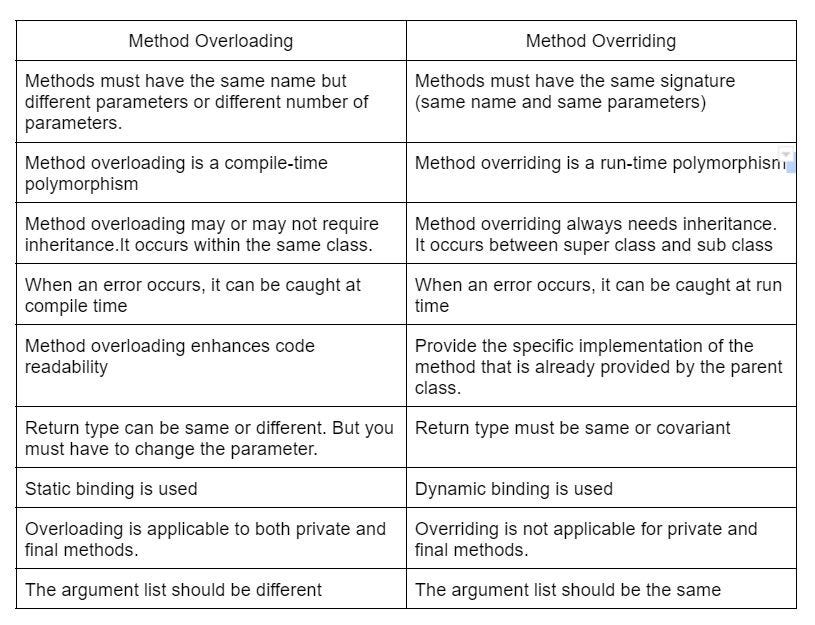
**package com.mercy.polymorphim;  
  
public class Client2 {  
 public static void main(String[] args) {  
 Animal1 a1 = new Animal1();  
 Animal1 a2 = new Dog1();  
 Animal1 a3 = new Cat1();  
  
 a1.makeSound();  
 a2.makeSound();  
 a3.makeSound();  
 }  
}**

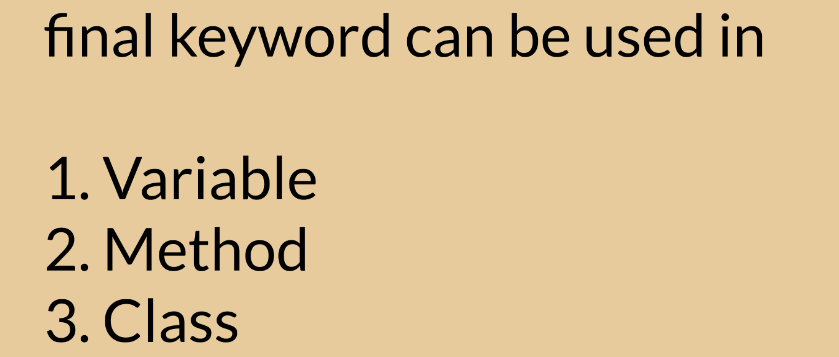
**package com.mercy.polymorphim;  
  
import java.util.List;  
  
public class Client3 {  
 public static void main(String[] args) {  
 Animal1 a1 = new Animal1();  
 Animal1 a2 = new Dog1();  
 Animal1 a3 = new Cat1();  
  
 List<Animal1> animalList = List.*of*(a1, a2, a3);  
  
 for(Animal1 animal : animalList){  
 animal.makeSound();  
 }  
 }  
}**

**package com.mercy.polymorphim;  
  
public class Client4 {  
 public static void process(Animal1 obj){  
 System.*out*.println("process(Animal1 obj)");  
 obj.makeSound();  
 }  
 public static void main(String[] args) {  
 Animal1 a1 = new Animal1();  
 Animal1 a2 = new Dog1();  
 Animal1 a3 = new Cat1();  
  
 *process*(a1);  
 *process*(a2);  
 *process*(a3);  
 }  
}**

Key Points:

* **Method Overloading**: Same method name, different parameter list (compile-time).
* **Method Overriding:** Same method signature, subclass provides a specific implementation (runtime).
* Polymorphism enables flexibility, code reusability, and dynamic method dispatch.





1. final variable value cannot be changed

2- final variable can be initialized only once

3- final variable cannot be re-initialized

* final with local variable
* final with instance variable
* final with static variable

**package com.mercy.finalkeyword;  
  
public class Client1 {  
 public static void main(String[] args) {  
 *// local variable* int x = 10;  
 System.*out*.println("x = "+x);  
  
 *// final local variable* final int y = 20;  
 System.*out*.println("y = "+y);  
  
 x = 30; *// this is allowed as x is not final variable  
// y = 30; // Cannot assign a value to final variable 'y'* }  
}**

**package com.mercy.finalkeyword;  
  
class Employee {  
  
}  
  
public class Client2 {  
 public static void main(String[] args) {  
 *// local variables* Employee e1 = new Employee();  
 Employee e2 = new Employee();  
 e1 = e2; *// allowed because e1 is not final variable* final Employee e3 = new Employee();  
*// e3 = e2; // Cannot assign a value to final variable 'e3'* }  
}**

**package com.mercy.finalkeyword;  
  
public class Client3 {  
 public static void main(String[] args) {  
 String name1 = "Angel";  
 name1 = "Mercy"; *// allowed because name1 is not final variable* final String name2 = "John";  
*// name2 = "Mathew"; // Cannot assign a value to final variable 'name2'* }  
}**

**package com.mercy.finalkeyword;  
  
public class Client4 {  
 public static void main(String[] args) {  
 int[] numberArray = new int[2]; *// numberArray is having size 2* numberArray = new int[5]; *// allowed because numberArray is not final variable* final int[] numberArray1 = new int[10];*// numberArray1 is having size 10  
// numberArray1 = new int[15];// Cannot assign a value to final variable 'numberArray1'* }  
}**

**package com.mercy.finalkeyword;  
public class Client5 {  
 *// instance variable* int x;  
  
 *// instance variable  
 // final int y; // Variable 'y' might not have been initialized* final int y = 10;  
}**

**package com.mercy.finalkeyword;  
  
public class Client6 {  
 *// final instance variable* final int x;  
  
 public Client6(int x) {  
 this.x = x;  
 }  
}**

**package com.mercy.finalkeyword;  
  
public class Client7 {  
 *// final instance variable* final int x = 10;  
  
 public Client7(int x) {  
*// this.x = x; // final variable cannot be re-initialized* }  
}**

**package com.mercy.finalkeyword;  
  
public class Client8 {  
 *// final instance variable* final int x;  
  
 *// Instance Initialization Block* {  
 x = 10;  
 }  
 {  
*// x = 10; // final variable cannot be re-initialized* }  
}**

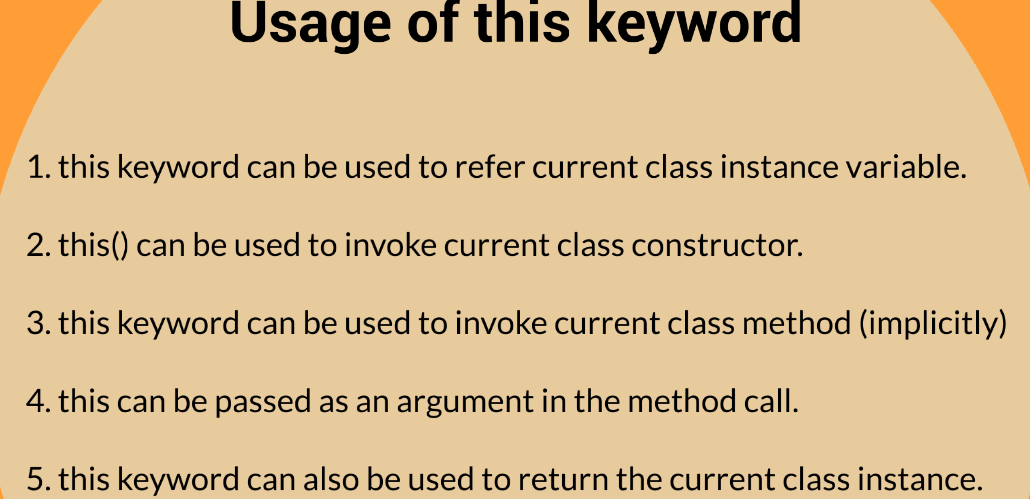
**package com.mercy.finalkeyword;  
  
public class Client9 {  
 final static int *COUNT* = 10;  
 final static int *COUNT\_ONE*;  
 *// SIB* static {  
 *COUNT\_ONE* = 100;  
 }  
}**

**if you declare a method as final, you cannot override it**

**package com.mercy.finalkeyword;  
  
public class Animal {  
 *// non-final method* public void makeSound(){  
 System.*out*.println("Some animal is making sound");  
 }  
  
 *// final method* final public void move(){  
 System.*out*.println("Some animal is moving");  
 }  
}  
  
class Dog extends Animal {  
 @Override  
 public void makeSound(){  
 System.*out*.println("Woof Woof");  
 }  
*// 'move()' cannot override 'move()' in 'com.mercy.finalkeyword.Animal';  
// overridden method is final  
// public void move(){  
// System.out.println("Dog is moving");  
// }*}**

**if you declare class as final, then you cannot extend it.**

**package com.mercy.finalkeyword;  
  
final public class Person {  
}  
  
*// Cannot inherit from final 'com.mercy.finalkeyword.Person'*class EmployeeNew extends Person {  
  
}**



**package com.mercy.thiskeyword;  
  
public class Employee {  
 *// instance variables* private int id;  
 private String name;  
  
 *// constructor* public Employee(int id, String name) {  
 id = id; *// local variable* name = name; *// local variable* }  
  
 @Override  
 public String toString() {  
 return "Employee{" +  
 "id=" + id +  
 ", name='" + name + '\'' +  
 '}';  
 }  
}**

**package com.mercy.thiskeyword;  
  
public class Employee1 {  
 *// instance variables* private int id;  
 private String name;  
  
 public Employee1(int id, String name){  
 this.id = id;  
 this.name = name;  
 }  
  
 @Override  
 public String toString() {  
 return "Employee1{" +  
 "id=" + id +  
 ", name='" + name + '\'' +  
 '}';  
 }  
}**

**package com.mercy.thiskeyword;  
  
public class Client1 {  
 public static void main(String[] args) {  
 Employee e1 = new Employee(100, "Mercy");  
 System.*out*.println(e1);  
 System.*out*.println("=======================");  
 Employee1 e2 = new Employee1(100, "Mercy");  
 System.*out*.println(e2);  
 System.*out*.println("=======================");  
 Employee2 e3 = new Employee2(100, "Mercy");  
 System.*out*.println(e3);  
 }  
}**

**this can be used to return current class object.**

**package com.mercy.thiskeyword;  
  
public class Employee4 {  
 *// instance variables* private int id;  
 private String name;  
  
 public Employee4(int id, String name) {  
 this.id = id;  
 this.name = name;  
 }  
  
 public void init(){  
 System.*out*.println("init() method begins");  
 process(); *// directly called since it is in instance method. compiler adds it***

**System.*out*.println("init() method ends");  
 }  
  
 public void process(){  
 System.*out*.println("process() method begins");  
 this.destroy(); *// called using this -> using this keyword is not mandatory* System.*out*.println("process() method ends");  
 }  
  
 public void destroy(){  
 System.*out*.println("destroy() method begins");  
 System.*out*.println("destroy() method ends");  
 }  
}  
  
class Client{  
 public static void main(String[] args) {  
 Employee4 employee4 = new Employee4(200,"Angel");  
 employee4.init(); *// called using employee4 reference variable* }  
}**

**this can be used as an argument to method call.**

**package com.mercy.thiskeyword;  
  
public class Employee5 {  
 *// instance variables* private int id;  
 private String name;  
  
 public Employee5(int id, String name) {  
 this.id = id;  
 this.name = name;  
 }  
  
 public Employee5 getInstance() {  
 System.*out*.println(toString());  
  
 this.id = 500;  
 this.name = "Mercy";  
 System.*out*.println(this);  
 System.*out*.println("=================");  
 updateInstance(this);  
  
 return this;  
 }  
  
 public void updateInstance(Employee5 employee5) {  
 employee5.name = "Mercy Technologies";  
 employee5.id = 5000;  
 }  
  
 @Override  
 public String toString() {  
 return "Employee5{" +  
 "id=" + id +  
 ", name='" + name + '\'' +  
 '}';  
 }  
}  
  
class Client5 {  
 public static void main(String[] args) {  
 Employee5 employee5 = new Employee5(300, "Angle");  
 System.*out*.println(employee5);  
 System.*out*.println("=================");  
 Employee5 empInstance = employee5.getInstance();  
 System.*out*.println(empInstance);  
 }  
}**

**Abstract Class**

**What is data abstraction?**

Data abstraction is the process of hiding implementation details and showing only essential information to the user.

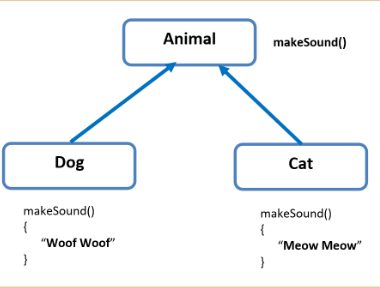
What is abstract class?

A class which is declared using “abstract” keyword is known as abstract class.

abstract class can have below things

* Abstract methods (methods without body)
* Non-abstract methods (regular methods with body)
* Both abstract methods & non-abstract methods

**Why do we need abstract class?**



Abstract Classes:

===========

1- id a method does not have body, then it should be declared as abstract.

using abstract keyword.

2- if a class contains even a single abstract method,

then it should be declared as abstract.

3- an abstract class can have more than one abstract methods.

4- abstract class can have abstract method & non-abstract methods as well.

5- for method that has implementation, we should have abstract keyword.

6- an abstract method shuld not have body.

7- abstract class cannot be INSTANTIATED!

in other words, we cannot create objects to abstract classes.

8- Illegal combination of modifiers 'abstract' and 'final'

9- Illegal combination of modifiers 'abstract' and 'private'

**package com.mercy.abst;  
  
public class Animal {  
 *// This implementation does not give any specific info* void makeSound() {  
 System.*out*.println("Animal making some sound!");  
 }  
}**

**package com.mercy.abst;  
  
public abstract class Animal1 {  
 *// abstract class can have more than one abstract method* abstract void makeSound();  
 abstract void move();  
 abstract boolean isMammal();  
}**

**package com.mercy.abst;  
  
public abstract class Animal3 {  
 *// abstract class can have more than one abstract method* abstract void makeSound();  
  
 abstract void move();  
  
 abstract boolean isMammal();  
  
 *// concrete method or non-abstract method* void findAge() {  
 System.*out*.println("find age based on dob");  
 }  
}**

**package com.mercy.abst;  
  
public class Client3 {  
 public static void main(String[] args) {  
 *// 'Animal3' is abstract; cannot be instantiated* Animal3 a3 = new Animal3();  
 }  
}**

**package com.mercy.abst;  
  
public class Client33 {  
 public static void main(String[] args) {  
 Dog3 d3 = new Dog3();  
 *// implemented in Dog3* d3.makeSound();  
 d3.move();  
 d3.isMammal();  
 *// from parent class* d3.findAge();  
 }  
}**

**package com.mercy.abst;  
  
public abstract class Animal4 {  
 *// abstract class can have more than one abstract method* abstract void makeSound();  
  
 abstract void move();  
  
 abstract boolean isMammal();  
  
 *// concrete method or non-abstract method* void findAge() {  
 System.*out*.println("find age based on dob");  
 }  
}**

**package com.mercy.abst;  
  
public abstract class Mammal4 extends Animal4{  
 @Override  
 boolean isMammal() {  
 System.*out*.println("This is Mammal!");  
 return true;  
 }  
}**

**package com.mercy.abst;  
  
public class Client4 {  
 public static void main(String[] args) {  
 Cat4 c4 = new Cat4();  
 *// from Animal4* c4.findAge();  
 *// from Mammal4* c4.isMammal();  
 *// from Cat4* c4.makeSound();  
 c4.move();  
 }  
}**

**package com.mercy.abst;  
  
public abstract class Animal5 {  
 *// Illegal combination of modifiers 'abstract' and 'private'* private abstract void makeSound();  
  
 abstract void move();  
  
 abstract boolean isMammal();  
  
 *// concrete method or non-abstract method* void findAge() {  
 System.*out*.println("find age based on dob");  
 }  
}**

**package com.mercy.abst;  
  
*// Illegal combination of modifiers 'abstract' and 'private'*private abstract class Animal6 {  
 *// Illegal combination of modifiers 'abstract' and 'final'* final abstract void makeSound();  
  
 abstract void move();  
  
 abstract boolean isMammal();  
  
 *// concrete method or non-abstract method* void findAge() {  
 System.*out*.println("find age based on dob");  
 }  
}**

1- private access modifier is not allowed with abstract

* default
* protected
* public

2- all above 3 access modifiers are allowed with abstract keyword

3- We can go for higher access level in the child class. in other words, we can increase the visibility.

**private < default < protected < public**

**parent** **child**

1. default -> default or protected or public
2. protected -> protected or public
3. public -> public

4- We cannot decrease the visibility. We get compile time error if we do so.

**package com.mercy.abst;  
  
public abstract class Animal7 {  
 *// default access modifier is allowed with abstract* abstract void makeSound();  
}**

**package com.mercy.abst;  
  
public abstract class Animal8 {  
 *// protected access modifier is allowed with abstract* protected abstract void makeSound();  
}**

**package com.mercy.abst;  
  
public class Dog88 extends Animal8{  
 @Override  
 protected void makeSound() { *// we cannot decrease the visibility* System.*out*.println("Woof Woof");  
 }  
}  
class Client88 {  
 public static void main(String[] args) {  
 Dog88 d88 = new Dog88();  
 d88.makeSound();  
 }  
}**

**package com.mercy.abst;  
  
public abstract class Animal9 {  
 *// public access modifier is allowed with abstract* public abstract void makeSound();  
}**

**package com.mercy.abst;  
  
public class Dog9 extends Animal9{  
 @Override  
 public void makeSound() {  
 System.*out*.println("Woof Woof");  
 }  
}**

**Interface**

Another way to achieve abstraction in Java, is with interfaces.

An interface is a completely "**abstract class**" that is used to group related methods with empty bodies:

* interface is used to achieve 100% abstraction in java
* interface can be a member of a .java file
* For interface also, after compilation, we get a .class file only (not .interface file)
* Modifier 'public' is redundant for interface members
* Modifier 'abstract' is redundant for interface methods
* All methods in interface are public and abstract by default
* We cannot have concrete methods in interface
* Constructor is not allowed in interface
* SIB is not allowed in interface
* We can have a class (inner class) in interface
* We can have multiple classes in interface
* We can have an abstract class in interface
* We can have multiple abstract classes in interface

**package com.mercy.interf;  
public interface Animal {  
 public abstract void makeSound();  
 public abstract void move();  
 public abstract void isMammal();  
}**

**package com.mercy.interf;  
public interface Animal1 {  
 *// Modifier 'public' is redundant for interface members* public abstract void makeSound();  
 *// Modifier 'abstract' is redundant for interface methods* abstract void move();  
 *// all methods in interface are public and abstract by default* void isMammal();  
}**

**package com.mercy.interf;  
public interface Animal3 {  
 *// concrete method, as it has a method body  
 // Interface abstract methods cannot have body* void makeSound() {  
 System.*out*.println("Some Animal making sound");  
 }  
}**

**package com.mercy.interf;  
public interface Animal4 {  
 *// constructor is not allowed in interface* Animal4() {  
  
 }  
 *// SIB is not allowed in interface* static {  
  
 }  
 *// IIB is not allowed in interface* {  
  
 }  
 *// We can have a class (inner class) in interface* class Dog {  
  
 }  
  
 *// We can have multiple classes in interface* class Cat {  
  
 }**

***// We can have an abstract class in interface* abstract class Animal100 {  
  
 }  
 *// We can have multiple abstract classes in interface* abstract class Animal200 {  
  
 }  
}**

**Interfaces part-2**

==============

1- One class can implement an interface

2- One class can extend another class

3- implements is denoted using dotted lines

4- extends is denoted using solid lines

5- We cannot instantiate an interface

6- We can create a reference variable of type interface

7- We can interchangably use intefaces and abstract classes.

8- We can define all the methods in the inteface and start adding the implementations in the relevant clsses.

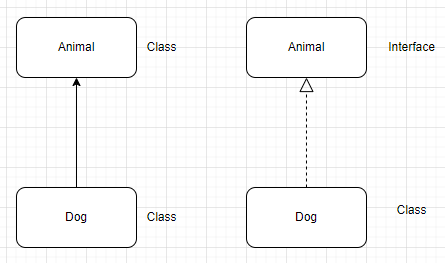
9- If we are unsure of the implementation, then we can leave it abstract and give the proper implementation in the child class.

10- We can use both extends and implements togather.

11- We can both extend and implement at the same time.

12- We can implement multiple interfaces (this is multiple inheritance in java)

13- We cannot achieve multiple inheritance using classes



**package com.mercy.interf;  
  
public interface Animal55 {  
 void makeSound();  
}**

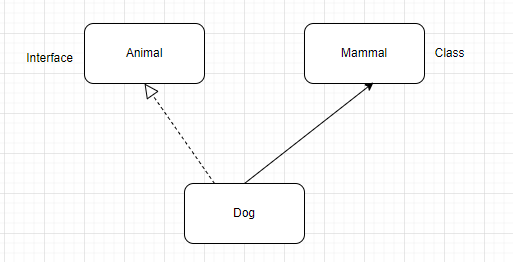
**package com.mercy.interf;  
  
public class Dog55 implements Animal55{  
 @Override  
 public void makeSound() {  
 System.*out*.println("Woof Woof");  
 }  
}**

**package com.mercy.interf;  
  
public class Client55 {  
 public static void main(String[] args) {  
 *// 'Animal55' is abstract; cannot be instantiated  
// Animal55 a55 = new Animal55();* Animal55 a55 = new Dog55();  
 a55.makeSound();  
 }  
}**

**package com.mercy.interf;  
  
public interface Animal66 {  
 void makeSound();  
 void move();  
 boolean isMammal();  
 boolean isReptile();  
}**

**package com.mercy.interf;  
  
public class Dog66 implements Animal66{  
 @Override  
 public void makeSound() {  
 System.*out*.println("Woof Woof");  
 }  
  
 @Override  
 public void move() {  
 System.*out*.println("Dog is moving using 4 legs");  
 }  
  
 @Override  
 public boolean isMammal() {  
 System.*out*.println("Dog is a Mammal");  
 return true;  
 }  
  
 @Override  
 public boolean isReptile() {  
 System.*out*.println("Dog is not a reptile");  
 return false;  
 }  
}**

**package com.mercy.interf;  
  
public class Client66 {  
 public static void main(String[] args) {  
 Animal66 a66 = new Dog66();  
 a66.makeSound();  
 a66.move();  
 a66.isMammal();  
 a66.isReptile();  
 }  
}**



**package com.mercy.interf;  
  
public interface Animal88 {  
 void makeSound();  
}**

**package com.mercy.interf;  
  
public abstract class Mammal88 {  
 abstract boolean isMammal();  
}**

**package com.mercy.interf;  
  
public class Dog88 extends Mammal88 implements Animal88{  
 @Override  
 public void makeSound() {  
  
 }  
  
 @Override  
 boolean isMammal() {  
 return false;  
 }  
}**

**package com.mercy.interf;  
  
public interface Animal99 {  
 void makeSound();  
}**

**package com.mercy.interf;  
  
public interface Animal999 {  
 void makeSound();  
}**

**package com.mercy.interf;  
  
public class Animal9\_9 {  
 void makeSound() {  
 System.*out*.println("Animal9\_9 ia making sound");  
 }  
}**

**package com.mercy.interf;  
  
public class Animal9\_9\_9 {  
 void makeSound() {  
 System.*out*.println("Animal9\_9\_9 ia making sound");  
 }  
}**