# Track 1

## Variable Types and Its size

| Type | Size (bits) | Minimum | Maximum | Examples |
|---|---|---|---|---|
| byte | 8 | $-2^7$ | $2^7-1$ | byte b = 100; |
| short | 16 | $-2^{15}$ | $2^{15}-1$ | short s = 30_000; |
| int | 32 | $-2^{31}$ | $2^{31}-1$ | int i = 100_000_000; |
| long | 64 | $-2^{63}$ | $2^{63}-1$ | long l = 100_000_000_000_000; |
| float | 32 | $-2^{-149}$ | $(2-2^{-23})\cdot2^{127}$ | float f = 1.456f; |
| double | 64 | $-2^{-1074}$ | $(2-2^{-52})\cdot2^{1023}$ | double f = 1.456789012345678; |
| char | 16 | 0 | $2^{16}-1$ | char c = 'c'; |
| boolean | 1 | – | – | boolean b = true; |

## Keywords in Java

| | | |
|---|---|---|
| 1. | **abstract** | Specifies that a class or method will be implemented later, in a subclass |
| 2. | **assert** | Assert describes a predicate placed in a java program to indicate that the developer thinks that the predicate is always true at that place. |
| 3. | **boolean** | A data type that can hold True and False values only |
| 4. | **break** | A control statement for breaking out of loops. |
| 5. | **byte** | A data type that can hold 8-bit data values |
| 6. | **case** | Used in switch statements to mark blocks of text |
| 7. | **catch** | Catches exceptions generated by try statements |
| 8. | **char** | A data type that can hold unsigned 16-bit Unicode characters |
| 9. | **class** | Declares a new class |
| 10. | **continue** | Sends control back outside a loop |

| | | |
|---|---|---|
| 11. | **default** | Specifies the default block of code in a switch statement |
| 12. | **do** | Starts a do-while loop |
| 13. | **double** | A data type that can hold 64-bit floating-point numbers |
| 14. | **else** | Indicates alternative branches in an if statement |
| 15. | **enum** | A Java keyword is used to declare an enumerated type. Enumerations extend the base class. |
| 16. | **extends** | Indicates that a class is derived from another class or interface |
| 17. | **final** | Indicates that a variable holds a constant value or that a method will not be overridden |
| 18. | **finally** | Indicates a block of code in a try-catch structure that will always be executed |
| 19. | **float** | A data type that holds a 32-bit floating-point number |
| 20. | **for** | Used to start a for loop |
| 21. | **if** | Tests a true/false expression and branches accordingly |
| 22. | **implements** | Specifies that a class implements an interface |
| 23. | **import** | References other classes |
| 24. | **instanceof** | Indicates whether an object is an instance of a specific class or implements an interface |
| 25. | **int** | A data type that can hold a 32-bit signed integer |
| 26. | **interface** | Declares an interface |
| 27. | **long** | A data type that holds a 64-bit integer |
| 28. | **native** | Specifies that a method is implemented with native (platform-specific) code |
| 29. | **new** | Creates new objects |
| 30. | **null** | This indicates that a reference does not refer to anything |
| 31. | **package** | Declares a Java package |
| 32. | **private** | An access specifier indicating that a method or variable may be accessed only in the class it's declared in |
| 33. | **protected** | An access specifier indicating that a method or variable may only be accessed in the class it's declared in (or a subclass of the class it's declared in or other classes in the same package) |
| 34. | **public** | An access specifier used for classes, interfaces, methods, and variables indicating that an item is accessible throughout the application (or where the class that defines it is accessible) |
| 35. | **return** | Sends control and possibly a return value back from a called method |
| 36. | **short** | A data type that can hold a 16-bit integer |
| 37 | **static** | Indicates that a variable or method is a class method (rather than being limited to one particular object) |
| 38. | **strictfp** | A Java keyword is used to restrict the precision and rounding of floating-point calculations to ensure portability. |
| 39. | **super** | Refers to a class's base class (used in a method or class constructor) |
| 40. | **switch** | A statement that executes code based on a test value |
| 41. | **synchronized** | Specifies critical sections or methods in multithreaded code |

| 42. | **this** | Refers to the current object in a method or constructor |
| 43. | **throw** | Creates an exception |
| 44. | **throws** | Indicates what exceptions may be thrown by a method |
| 45. | **transient** | Specifies that a variable is not part of an object's persistent state |
| 46. | **try** | Starts a block of code that will be tested for exceptions |
| 47. | **void** | Specifies that a method does not have a return value |
| 48. | **volatile** | This indicates that a variable may change asynchronously |
| 49. | **while** | Starts a while loop |

| 0 | <NUL> | 32 | <SPC> | 64 | @ | 96 | ` | 128 | Ä | 160 | † | 192 | ¿ | 224 | ‡ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | <SOH> | 33 | ! | 65 | A | 97 | a | 129 | Å | 161 | ° | 193 | i | 225 | · |
| 2 | <STX> | 34 | " | 66 | B | 98 | b | 130 | Ç | 162 | ¢ | 194 | ¬ | 226 | , |
| 3 | <ETX> | 35 | # | 67 | C | 99 | c | 131 | É | 163 | £ | 195 | √ | 227 | „ |
| 4 | <EOT> | 36 | $ | 68 | D | 100 | d | 132 | Ñ | 164 | § | 196 | ƒ | 228 | ‰ |
| 5 | <ENQ> | 37 | % | 69 | E | 101 | e | 133 | Ö | 165 | • | 197 | ≈ | 229 | Â |
| 6 | <ACK> | 38 | & | 70 | F | 102 | f | 134 | Ü | 166 | ¶ | 198 | Δ | 230 | Ê |
| 7 | <BEL> | 39 | ' | 71 | G | 103 | g | 135 | á | 167 | ß | 199 | « | 231 | Á |
| 8 | <BS> | 40 | ( | 72 | H | 104 | h | 136 | à | 168 | ® | 200 | » | 232 | Ë |
| 9 | <TAB> | 41 | ) | 73 | I | 105 | i | 137 | â | 169 | © | 201 | … | 233 | È |
| 10 | <LF> | 42 | * | 74 | J | 106 | j | 138 | ä | 170 | ™ | 202 | | 234 | Í |
| 11 | <VT> | 43 | + | 75 | K | 107 | k | 139 | ã | 171 | ´ | 203 | À | 235 | Î |
| 12 | <FF> | 44 | , | 76 | L | 108 | l | 140 | å | 172 | ¨ | 204 | Ã | 236 | Ï |
| 13 | <CR> | 45 | - | 77 | M | 109 | m | 141 | ç | 173 | ≠ | 205 | Õ | 237 | Ì |
| 14 | <SO> | 46 | . | 78 | N | 110 | n | 142 | é | 174 | Æ | 206 | Œ | 238 | Ó |
| 15 | <SI> | 47 | / | 79 | O | 111 | o | 143 | è | 175 | Ø | 207 | œ | 239 | Ô |
| 16 | <DLE> | 48 | 0 | 80 | P | 112 | p | 144 | ê | 176 | ∞ | 208 | – | 240 |  |
| 17 | <DC1> | 49 | 1 | 81 | Q | 113 | q | 145 | ë | 177 | ± | 209 | — | 241 | Ò |
| 18 | <DC2> | 50 | 2 | 82 | R | 114 | r | 146 | í | 178 | ≤ | 210 | " | 242 | Ú |
| 19 | <DC3> | 51 | 3 | 83 | S | 115 | s | 147 | ì | 179 | ≥ | 211 | " | 243 | Û |
| 20 | <DC4> | 52 | 4 | 84 | T | 116 | t | 148 | î | 180 | ¥ | 212 | ` | 244 | Ù |
| 21 | <NAK> | 53 | 5 | 85 | U | 117 | u | 149 | ï | 181 | µ | 213 | ' | 245 | ı |
| 22 | <SYN | 54 | 6 | 86 | V | 118 | v | 150 | ñ | 182 | ∂ | 214 | ÷ | 246 | ^ |
| 23 | <ETB> | 55 | 7 | 87 | W | 119 | w | 151 | ó | 183 | Σ | 215 | ◊ | 247 | ~ |
| 24 | <CAN> | 56 | 8 | 88 | X | 120 | x | 152 | ò | 184 | ∏ | 216 | ÿ | 248 | ‾ |
| 25 | <EM> | 57 | 9 | 89 | Y | 121 | y | 153 | ô | 185 | π | 217 | Ÿ | 249 | ˘ |
| 26 | <SUB> | 58 | : | 90 | Z | 122 | z | 154 | ö | 186 | ∫ | 218 | / | 250 | ˙ |
| 27 | <ESC> | 59 | ; | 91 | [ | 123 | { | 155 | õ | 187 | ª | 219 | € | 251 | ° |
| 28 | <FS> | 60 | < | 92 | \ | 124 | \| | 156 | ú | 188 | º | 220 | ‹ | 252 | ¸ |
| 29 | <GS> | 61 | = | 93 | ] | 125 | } | 157 | ù | 189 | Ω | 221 | › | 253 | ˝ |
| 30 | <RS> | 62 | > | 94 | ^ | 126 | ~ | 158 | û | 190 | æ | 222 | fi | 254 | ˛ |
| 31 | <US> | 63 | ? | 95 | _ | 127 | <DEL> | 159 | ü | 191 | ø | 223 | fl | 255 | ˇ |

*Figure 1* Ascii Chart

| **Editors** | **IDE** |
|---|---|
| <ul><li>Notepad</li><li>Notepad ++</li><li>Visual Code</li><li>Sublime Text</li></ul> | <ul><li>Eclipse</li><li>Intelli J</li><li>NetBeans</li></ul> |

JVM, JRE and JDK are platform **dependent** because the configuration of each OS differs. But Java is platform-independent.

1. **Java Development Kit (JDK**) is a software development environment used for developing Java applications and applets. It includes the Java Runtime Environment (JRE), an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), and other tools needed in Java development.

JDK usage [Build, Compile, and Execute Application]

Java follows WORA (Write once read anywhere)

2. **JRE** (Java Runtime Environment) is an installation package that provides an environment to only run (not develop) the java program (or application) onto your machine. JRE is only used by those who only want to run Java programs that are end-users of your system.

3. **JVM** (Java Virtual Machine) is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for executing the java program line by line, hence it is also known as an interpreter.

*Installation Of JDK*
1. Download jdk8
2. Set Environment Path Variables
3. Test the installation using the following commands

```
C:\Users\suman>javac -version
javac 17.0.2
C:\Users\suman>java -version
java version "17.0.2" 2022-01-18 LTS
Java(TM) SE Runtime Environment (build 17.0.2+8-LTS-86)
Java HotSpot(TM) 64-Bit Server VM (build 17.0.2+8-LTS-86, mixed mode, sharing)
C:\Users\suman>
```

*Not the above result is for JDK 17, JDK 8 result might differ*

A computer only understands Binary Numbers 0, 1

Steps to Execute JAVA Code

1. Write your Java code in a .java file [Source Code]
2. The compilation (Conversion of a Source code to a more machine-friendly language)
   The compilation will give you a .class file (.class file is also known as *ByteCode*)
3. Execution
   Run the compiled .class file using the following commands.

```
C:\Users\suman\Downloads\Y Hills\Source Code\Basics>javac Main.java

C:\Users\suman\Downloads\Y Hills\Source Code\Basics>java Main
Light Speed2.9979445867687994E8
Light Speed@
```

**RULES**

- You must have to obey
- Apart from imports, annotations and package names, all will reside inside a class
- The name of the file should match the class name

**Convention [Good Habits]**

You should follow

- Class Names -> Use PascalCase                                    KushankJain
- Variable and Method[function] names -> we use camelCase          kushankJain

## Packages
### Package Intro

A java package is a group of similar types of classes, interfaces and sub-packages.

Advantage of Java Packages

- Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- Java package provides access protection.
- Java package removes naming collision.

### Naming Convention

Say Website is: www.jdk8.com

So package name will start as **com.jdk8.####**

Internally it will create a nested folder structure like

- ➢ com 📁
  - ➢ jdk8 📁
    - ➢ #### 📁

# Loops

## Scope Of Variables

The reach of a variable will depend on where it is declared.

## Continue and Break Statements

- Continue
- Break

The above two keywords can be used to further control the loops

## While Loop

While loop can be also used for looping

- do while
- while

*The basic difference between do-while and while loop is that do while will at least execute for one time.*

# Track 2

## Wrapper Classes

### Intro
The Wrapper class in Java provides the mechanism to convert primitive into object and object into primitive.

### Autoboxing and Unboxing
We also have ==autoboxing== and ==unboxing== features that convert primitives into objects and objects into primitives automatically. The automatic conversion of a primitive into an object is known as autoboxing and vice-versa unboxing.

```
Integer num = 9;
Character ch = 'c';
```

**WrapperClass.java**

```java
package com.jdk8.oop.basics;

public class WrapperClass {

        public static void main(String[] args) {

                    String name = "rishav";
//                  Primitive Variables
                    int number = 10;
                    boolean isSad = false;
//                  Wrapping int to an Integer Object
                    Integer integer = new Integer(10);

//                  Wrapping boolean  to an Boolean Object
                    Boolean boolean1 = new Boolean(true);

                    byte by = 8;
                    int p = by;

                    // Generally You cannot fit a higher space data in lower space [Bucket - Mug
                    // Analogy]
//                  int m = 8;
//                  byte c=m;

//                  Here we have exception as long[64bit] is getting stored in float[32 bit],
//                  The reason is that both have different storing algorithms, and floating
numbers is  getting stored using higher efficiency IEEE algo
                    long lon = Long.MAX_VALUE;
                    float fl = lon;
                    System.out.println(fl);

                    byte b = 10;
                    short s = 20;
                    int i = 30;
                    long l = 40;
                    float f = 50.0F;
                    double d = 60.0D;
                    char c = 'a';
                    boolean b2 = true;

                    // Autoboxing: Converting primitives into objects
                    // With this mechanism So no need of "new" keyword
                    Byte byteobj = b;
                    Short shortobj = s;
                    Integer intobj = i;
                    Long longobj = l;
                    Float floatobj = f;
                    Double doubleobj = d;
                    Character charobj = c;
                    Boolean boolobj = b2;

                    // Unboxing: Converting Objects to Primitives Automatically ,again no use of
                    // "new" keyword
                    byte bytevalue = byteobj;
                    short shortvalue = shortobj;
                    int intvalue = intobj;
                    long longvalue = longobj;
                    float floatvalue = floatobj;
                    double doublevalue = doubleobj;
                    char charvalue = charobj;
                    boolean boolvalue = boolobj;

        }

}
```

# OOP [Object Orientation Programming] Intro

## Intro

Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic. An object can be defined as a data field that has unique attributes and behaviour.

Say a **mixer grinder**

1. Its Properties [has]
    a. Wattage
    b. Colour
    c. Speed
2. Its Functionality [does]
    a. grind()
    b. blend()
    c. juiceOut()

**We use classes in OOP,**

Classes are just like Formula/Recipe or a prototype for an Object.

Think like to make a Cake we need a Recipe

Recipe ---> Cake
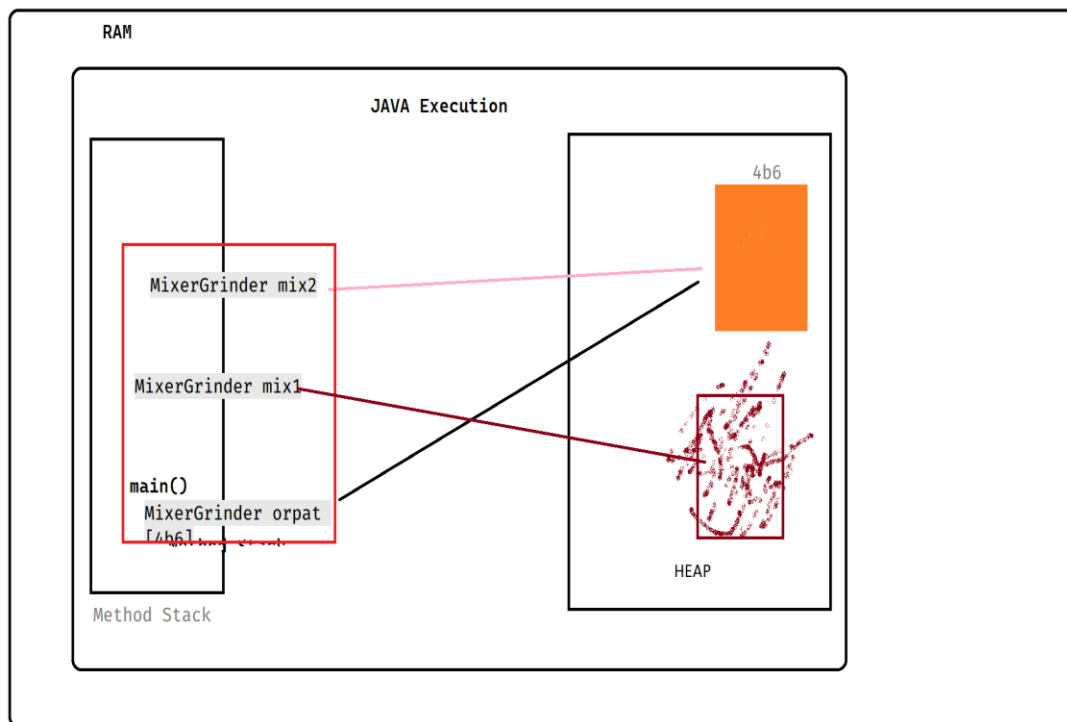
*Figure 2* OOP Memory Discussion

**MixerGrinder.java**

```java
package com.jdk8.oop.basics;

public class MixerGrinder {

        //Instance variable/Object Variables
        int wattage;
        String color;
        int speed;

        void grind(){
                System.out.println("GRINDING");
        }

        void mixing(){
                System.out.println("MIXING");
        }

        void blending(){
                System.out.println("BLENDING");
        }

        void readOutSpecs() {
                System.out.println("Wattage -> "+wattage);
                System.out.println("Speed -> "+speed);
                System.out.println("Color -> "+color);

        }

}
```

**MixerBuy.java**

```java
package com.jdk8.oop.basics;

public class MixerBuy {

        public static void main(String[] args) {
                // TODO Auto-generated method stub

                MixerGrinder orpat =new MixerGrinder();//orpat as a Object Reference
                //MixerGrinder()
                System.out.println(orpat);//By default it will print the memory address



                orpat.readOutSpecs();// preint the defaulyt values


//              Default Values
//              integer --> 0
//              float/double -> 0.0
//              Objetct refer nce - null
//              String - null
//              char -> null char
//              boolean --> false

                orpat.color="Pink";
                orpat.speed=20000;
                orpat.wattage=240;

                orpat.readOutSpecs();



                System.out.println("\n\n\n\n mix1");
                MixerGrinder mix1;
                mix1 = new MixerGrinder();
                System.out.println(mix1);
                mix1.readOutSpecs();


                MixerGrinder mix2= orpat;
                System.out.println(mix2);
                mix2.readOutSpecs();

                orpat.readOutSpecs();
//              mix1.readOutSpecs();

                mix1=null;
                orpat=null;

        }

}
```

## Garbage and Garbage Collector

Garbage collection in Java is the process by which Java programs perform automatic memory management. Java programs compile to bytecode that can be run on a Java Virtual Machine, or JVM for short. When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory.

Java garbage collection is an automatic process. Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects. An in-use object, or a referenced object, means that some part of your program still maintains a pointer to that object. An unused or unreferenced object is no longer referenced by any part of your program. So, the memory used by an unreferenced object can be reclaimed. The programmer does not need to mark objects to be deleted explicitly. The garbage collection implementation lives in the JVM.
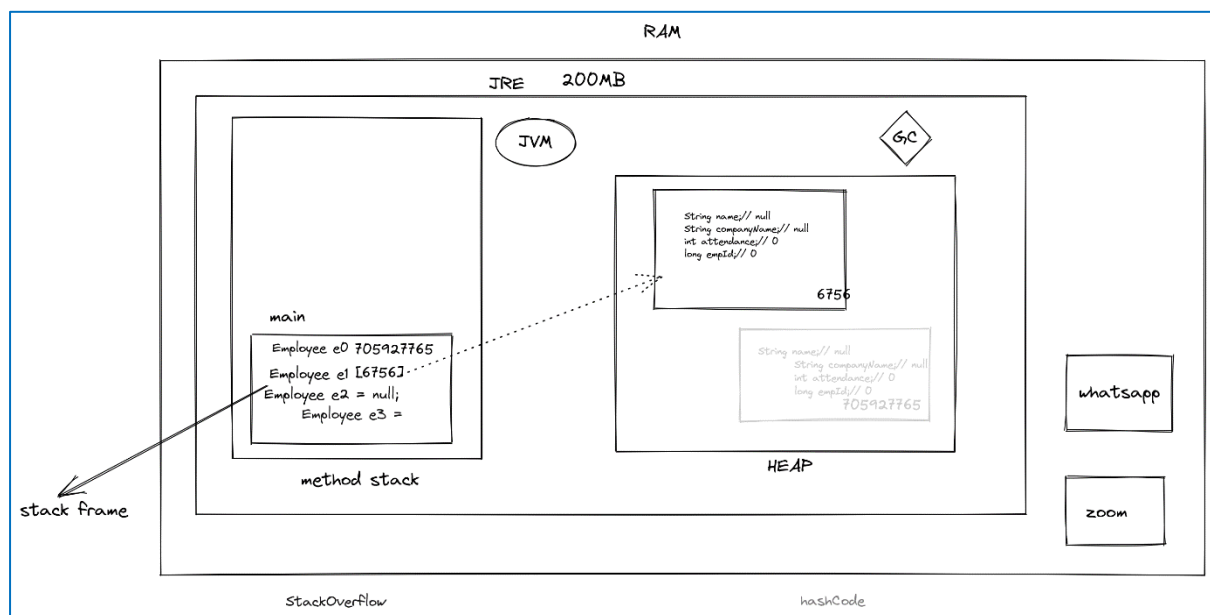


*Figure 3* Memory Working in Java

# Scanner Class

Scanner is a class in java.util package used for obtaining the input of the primitive types like int, double, etc. and strings. It is the easiest way to read input in a Java program.

**ScannerInput.java**

```java
package com.jdk8.basics;

import java.util.Scanner;

public class ScannerInput {

        public static void main(String[] args) {

//              Scanner Class [Inbuilt Class for taking input]
                Scanner sc = new Scanner(System.in);

                System.out.println("Enter your Age");// this will go to next line after
printing
                System.out.print("Enter your name");// this will not change the line after
printing
                System.out.println();// blank empty line
//              System.out.print();//Error

//              String next = sc.next();//inputs only one word
                String next = sc.nextLine();// inputs whole sentence
                System.out.println(next);

                float height = sc.nextFloat();
                System.out.println("Height ~> " + height);
                System.out.println("Enter a number to be squared");
                int integer = sc.nextInt();
                int result = findSqr(integer);
                System.out.println(result);

                sc.close();// closing the Scanner resource

        }

        static int findSqr(int x) {// formal parameters
                int result = x * x;
                return result;
        }

}
```

**ScannerInputCornerCase.java**

```java
package com.jdk8.basics;

import java.util.Scanner;

public class ScannerInputCornerCase {

        public static void main(String[] args) {

//              Scanner Class [Inbuilt]
                System.out.println("START");
                Scanner sc = new Scanner(System.in);
```

```
                System.out.println("Enter the age");
                int age = sc.nextInt();
                System.out.println("AGE -> " + age);

                System.out.println("Enter the name");
                sc.nextLine();// Flushing
                String name = sc.nextLine();
                System.out.println("NAME -> " + name);

                System.out.println("END");

        }

}
```

# Track 3

## Access Modifier

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.
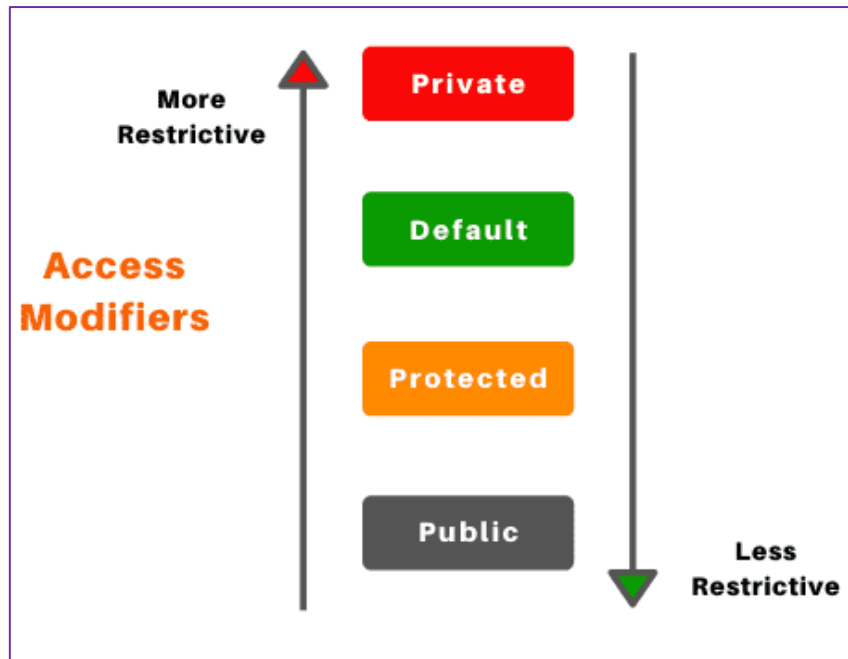
| Modifier | Class | Class Variables | Methods | Method Variables |
|---|---|---|---|---|
| public | ✓ | ✓ | ✓ | |
| private | | ✓ | ✓ | |
| protected | | ✓ | ✓ | |
| default | ✓ | ✓ | ✓ | |
| final | ✓ | ✓ | ✓ | ✓ |
| abstract | ✓ | | ✓ | |
| strictfp | ✓ | | ✓ | |
| transient | | ✓ | | |
| synchronized | | | ✓ | |
| native | | | ✓ | |
| volatile | | ✓ | | |
| static | | ✓ | ✓ | |

| Access Modifier | within class | within package | outside package by subclass only | outside package |
|---|---|---|---|---|
| Private | Y | N | N | N |

| | | | | |
|---|---|---|---|---|
| Default | Y | Y | N | N |
| Protected | Y | Y | Y | N |
| Public | Y | Y | Y | Y |

## Access Specifiers in Java

| | | public | private | protected | default |
|---|---|---|---|---|---|
| Same Package | Class | YES | YES | YES | YES |
| | Sub class | YES | NO | YES | YES |
| | Non sub class | YES | NO | YES | YES |
| Different Package | Sub class | YES | NO | YES | NO |
| | Non sub class | YES | NO | NO | NO |

| Access Specifier ▶ Item ▼ | Default | Public | Protected | Private |
|---|---|---|---|---|
| Class | Yes | Yes | No | No |
| Inner Class | Yes | Yes | Yes | Yes |
| Interface | Yes | Yes | No | No |
| Interface Inside Class | Yes | Yes | Yes | Yes |
| enum | Yes | Yes | No | No |
| enum Inside Class | Yes | Yes | Yes | Yes |
| enum inside Interface | Yes | No | No | No |
| Constructor | Yes | Yes | Yes | Yes |
| methods & data inside class | Yes | Yes | Yes | Yes |
| methods & data inside Interface | Yes | No | No | No |

Access control for members of class and interface in java

| Access Specifier / Accessibility Location | Same Class | Same Package | | Other Package | |
|---|---|---|---|---|---|
| | | Child class | Non-child class | Child class | Non-child class |
| Public | Yes | Yes | Yes | Yes | Yes |
| Protected | Yes | Yes | Yes | Yes | No |
| Default | Yes | Yes | Yes | No | No |
| Private | Yes | No | No | No | No |

## Polymorphism

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

## Method Overloading

Same method name but different functionality.

If a class has multiple methods having same name but different in parameters, it is known as Method Overloading.

If we have to perform only one operation, having same name of the methods increases the readability of the program

Rules

- No of parameter
- Type of paramerter
- Position of paramerter
- 

| Method Overloading | Method Overriding |
|---|---|
| Method overloading is used *to increase the readability* of the program. | Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class. |
| Method overloading is performed *within class*. | Method overriding occurs *in two classes* that have IS-A (inheritance) relationship. |
| In case of method overloading, *parameter must be different*. | In case of method overriding, *parameter must be same*. |
| Method overloading is the example of compile time polymorphism. | Method overriding is the example of run time polymorphism. |
| In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter. | *Return type must be same or covariant* in method overriding. |

## Abstraction

**Abstraction** is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets us focus on what the object does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

- Abstract class (0 to 100%)
- Interface (100%)

## Abstract Class

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

1. An abstract class must be declared with an abstract keyword.
2. It can have abstract and non-abstract methods.
3. It cannot be directly instantiated but we can use anonymous inner classes to instantiate and provide the body for abstract methods on the fly.
4. It can have constructors and static methods also.
5. It can have final methods which will force the subclass not to change the body of the method.

## Object Class

The Object class is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, know as upcasting.

The Object class provides many methods. They are as follows:

*Methods of Object class*

| Method | Description |
|---|---|
| public final Class getClass() | returns the Class class object of this object. The Class class can further be used to get the metadata of this class. |

| | |
|---|---|
| public int hashCode() | returns the hashcode number for this object. |
| public boolean equals(Object obj) | compares the given object to this object. |
| protected Object clone() throws CloneNotSupportedException | creates and returns the exact copy (clone) of this object. |
| public String toString() | returns the string representation of this object. |
| public final void notify() | wakes up single thread, waiting on this object's monitor. |
| public final void notifyAll() | wakes up all the threads, waiting on this object's monitor. |
| public final void wait(long timeout)throws InterruptedException | causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method). |
| public final void wait(long timeout,int nanos)throws InterruptedException | causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method). |
| public final void wait()throws InterruptedException | causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method). |
| protected void finalize()throws Throwable | is invoked by the garbage collector before object is being garbage collected. |

## CASTING

- // UPCASTING is implicit in nature
- // DOWNCASTING is explicit in nature