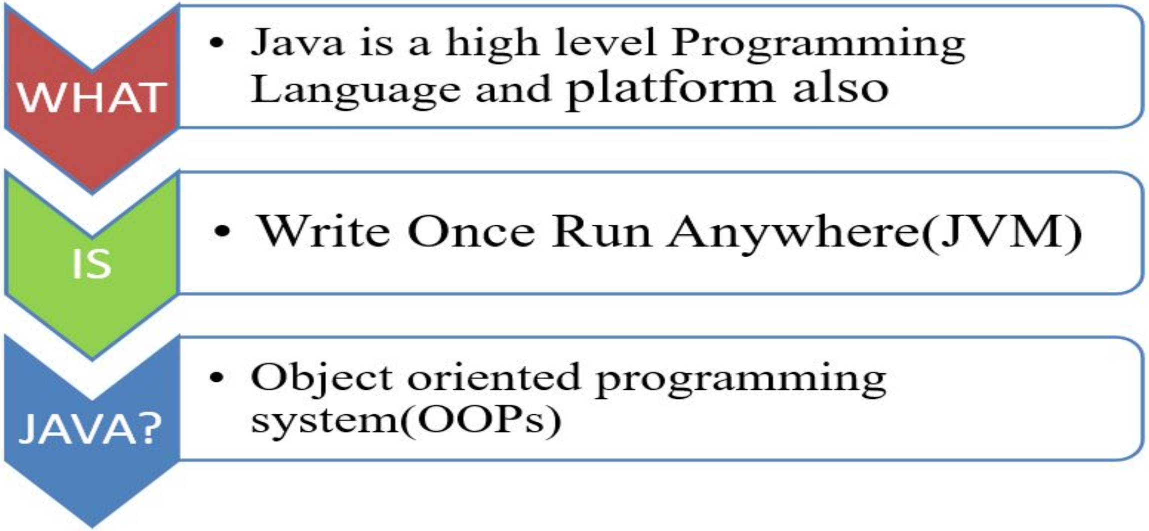
eTHNUS™

Explore | Expand | Enrich

<Codemithra />™

FUNDAMENTALS OF PROGRAMMING

Introduction Of Java



WHAT IS JAVA?

- Java is a high level Programming Language and platform also
- Write Once Run Anywhere(JVM)
- Object oriented programming system(OOPs)

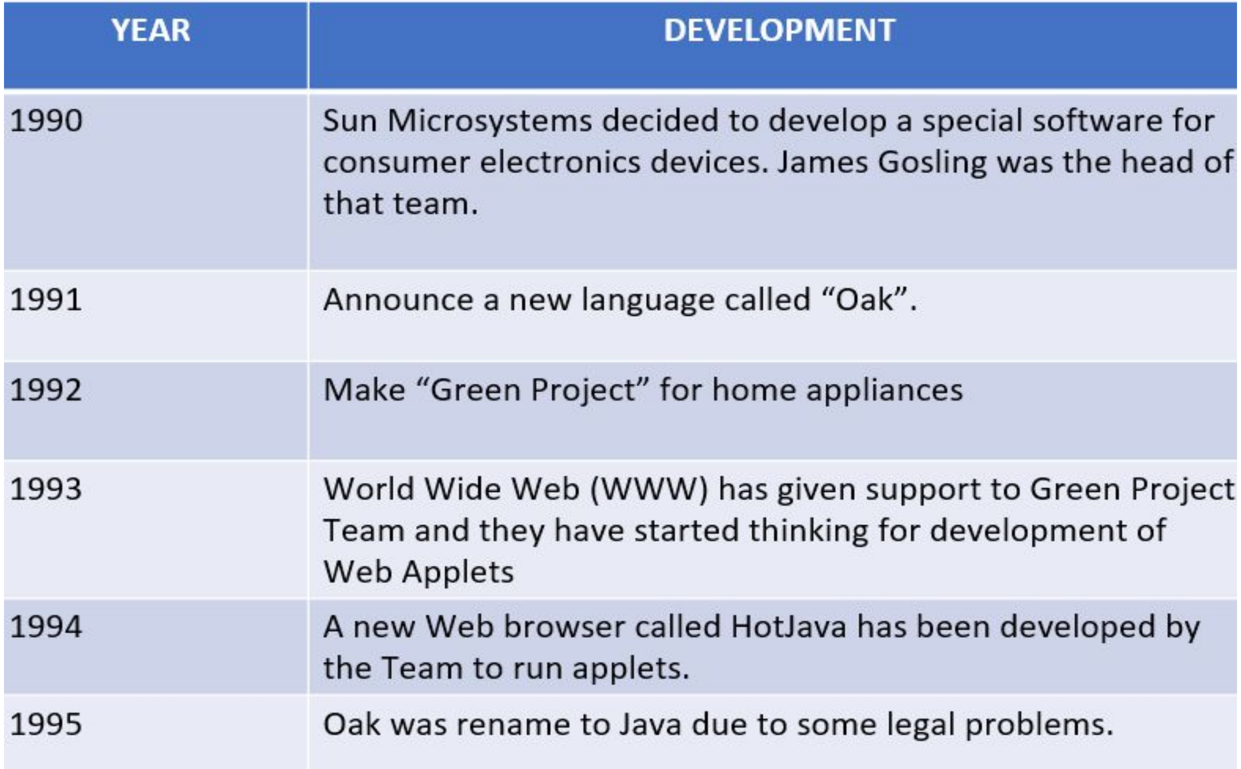
Where We Used

- Desktop Applications
- Mobile Applications
- Enterprise Applications
- Web-based Applications
- Gaming Applications

Where We Used

- Desktop Applications
- Mobile Applications
- Enterprise Applications
- Web-based Applications
- Gaming Applications

History

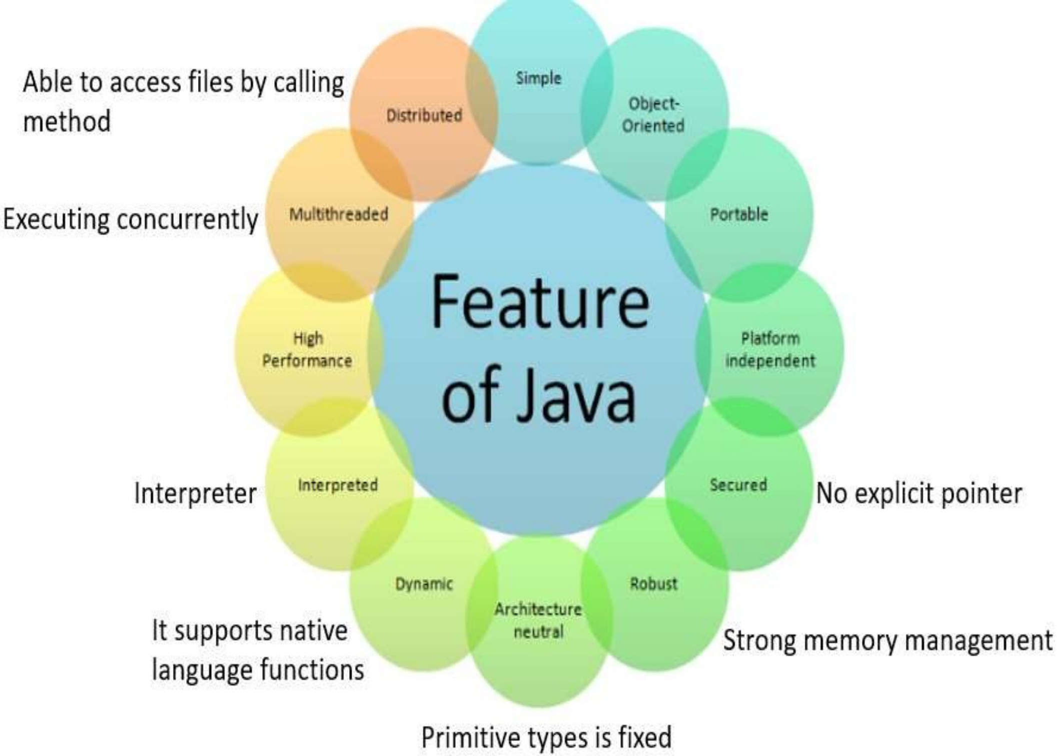


YEAR	DEVELOPMENT
1990	Sun Microsystems decided to develop a special software for consumer electronics devices. James Gosling was the head of that team.
1991	Announce a new language called “Oak”.
1992	Make “Green Project” for home appliances
1993	World Wide Web (WWW) has given support to Green Project Team and they have started thinking for development of Web Applets
1994	A new Web browser called HotJava has been developed by the Team to run applets.
1995	Oak was rename to Java due to some legal problems.

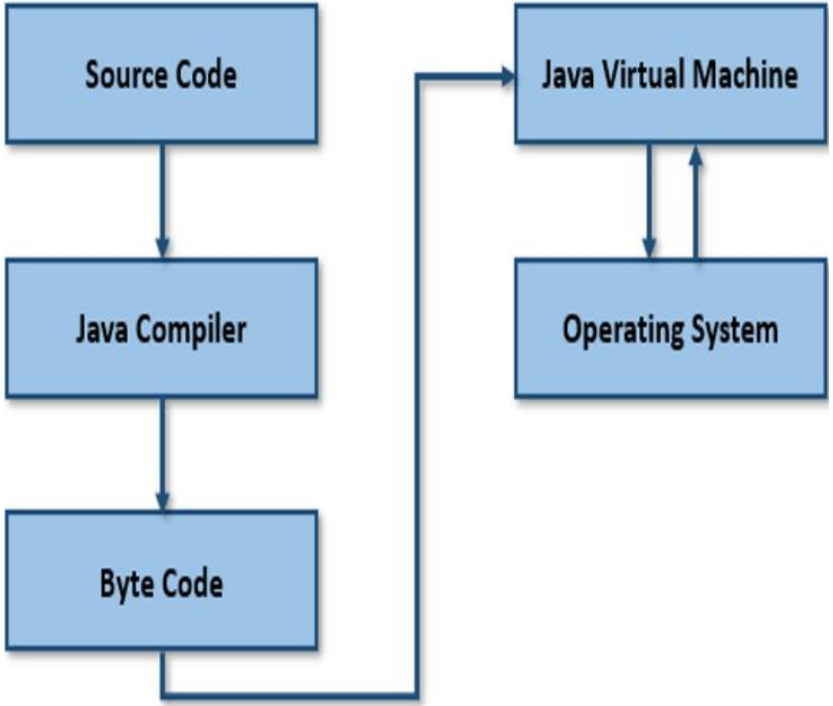
Latest Version

- JDK developed from 1995 onwards
- Currently we are using Java SE 19.0.1 released by Java SE Platform
- Released in September,2022
- May be March,2023, Java 20 will follow

Features


Feature of Java
Simple
Object-Oriented
Portable
Platform independent
Secured
Robust
Architecture neutral
Dynamic
Interpreted
High Performance
Multithreaded
Distributed
Able to access files by calling method
Executing concurrently
Interpreter
It supports native language functions
Primitive types is fixed
Strong memory management
No explicit pointer

Structure

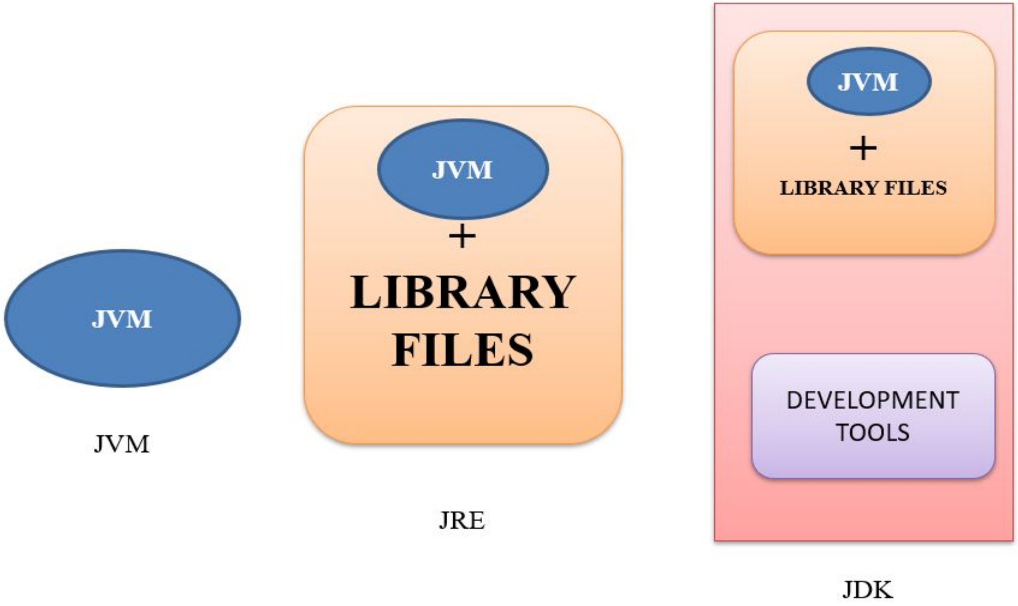

Source Code
Java Compiler
Byte Code
Java Virtual Machine
Operating System

Components

There are three main components of Java language:

- JVM(java Virtual Machine)
- JRE(Java Runtime Environment)
- JDK(Java Development Kit)

JVM,JRE,JDK


JVM
JVM
JVM
+
LIBRARY FILES
JRE
JVM
+
LIBRARY FILES
DEVELOPMENT TOOLS
JDK

JVM

JVM interprets the byte code into machine code which is executed

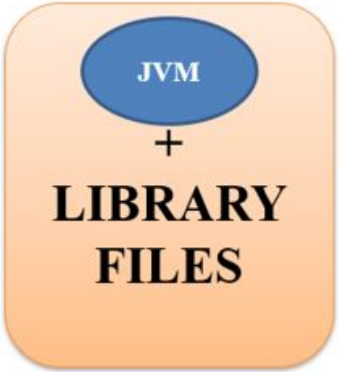
in the machine in which the Java program runs. Virtual manner.

Platform independent.

The JVM performs following main tasks:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JRE



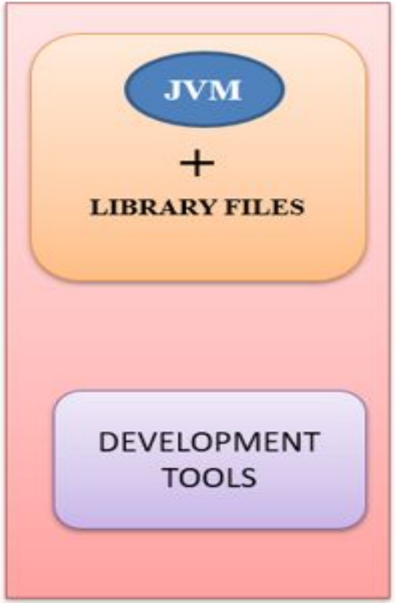
JVM

+

LIBRARY FILES

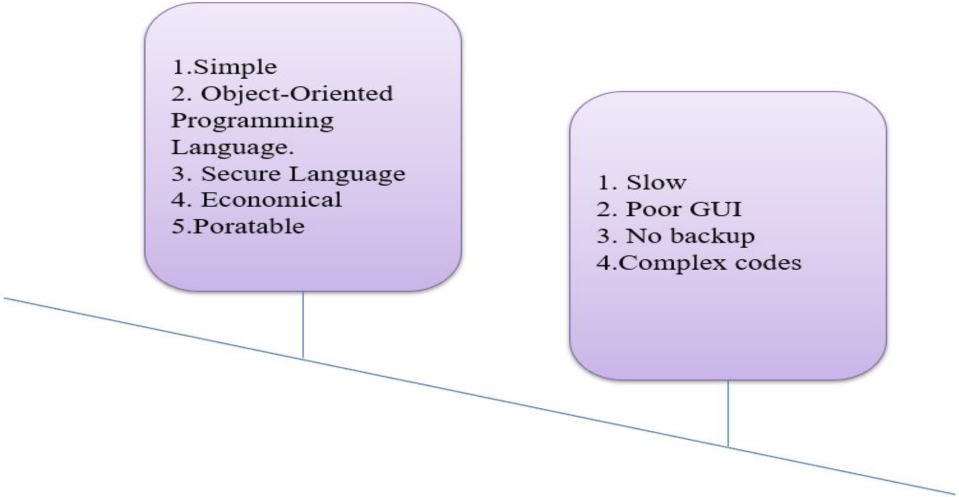
- It is used to provide runtime environment. It is the implementation of JVM.
- It physically exists.
- It contains set of libraries + other files that JVM uses at runtime.
- Must need to run a program

JDK



- The Java Development Kit (JDK) is physically exists.
- It contains JRE and several development tools, accompanied with another tool.
- Tools examples
  - Java compiler
  - Java doc
  - Applet viewer

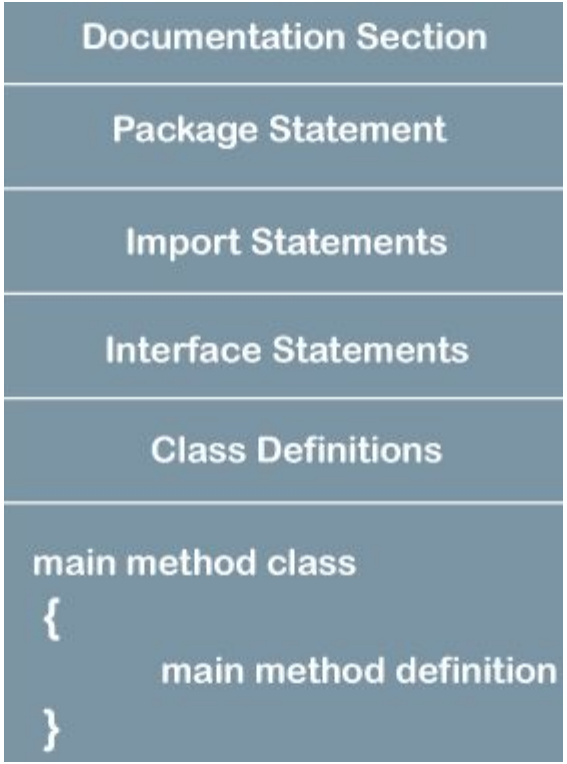
Pros and Cons



1.Simple
2. Object-Oriented Programming Language.
3. Secure Language
4. Economical
5.Poratable

1. Slow
2. Poor GUI
3. No backup
4.Complex codes

structure



Documentation Section

Package Statement

Import Statements

Interface Statements

Class Definitions

```
main method class
{
        main method definition
}
```

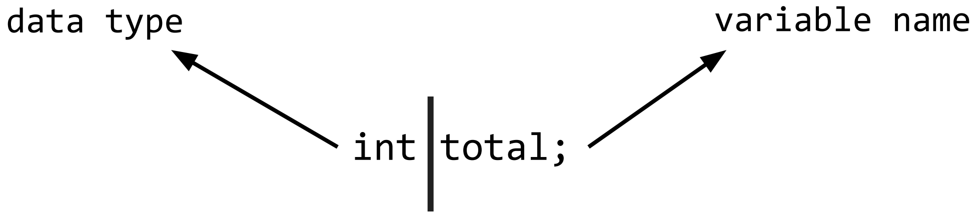
Data types

- Data is the information that a program has to work with.
- Data is of different types. The type of a piece of data tells Java what can be done with it, and how much memory needs to be put aside for it.

- When we create a variable in Java, we need to specify:
  - the type of the value we want to put in there
  - the name we will use for that variable.

Data types

- A variable must be declared, specifying the variable's name and the type of information that will be held in it



data type

variable name

int total;

int count, temp, result;

Multiple variables can be created in one declaration

int count=1, temp=0;

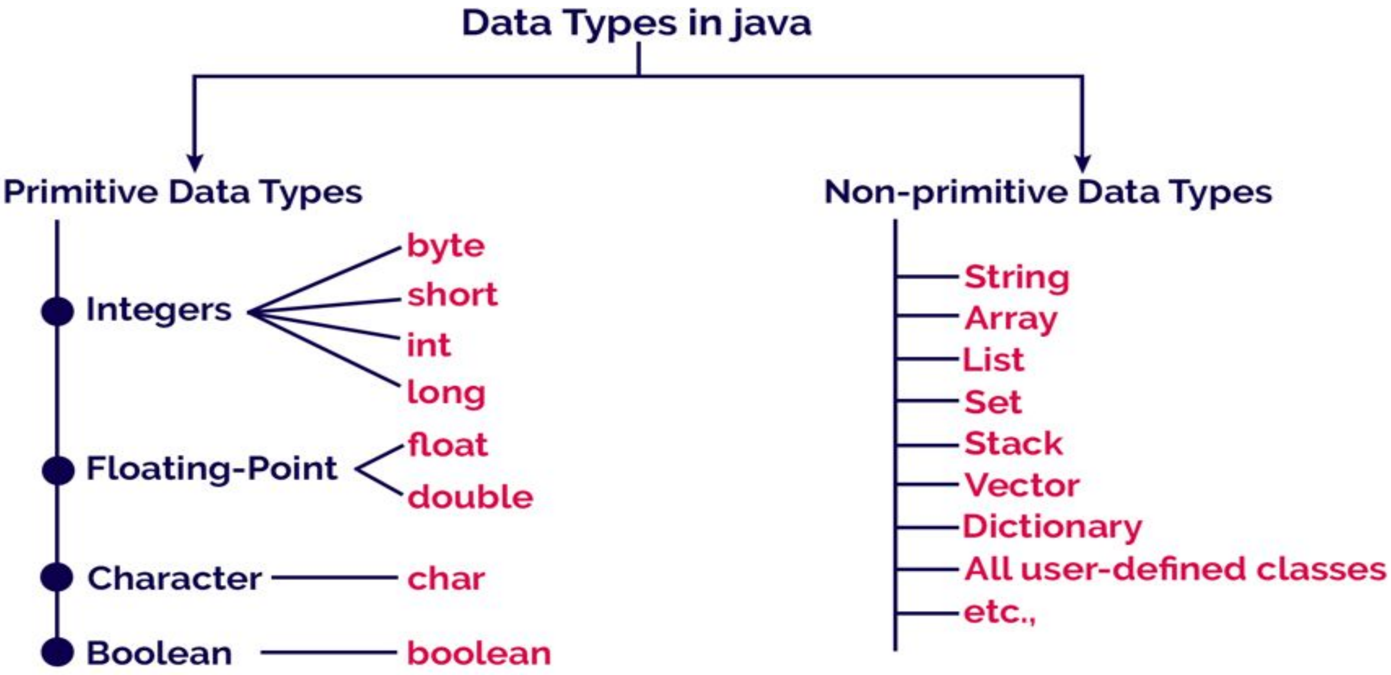
Variables can also be given initial values

Data types

data types are classified into two types and they are as follows.

- Primitive Data Types
- Non-primitive Data Types

Data types



Data Types in java

- Primitive Data Types
  - Integers
    - byte
    - short
    - int
    - long
  - Floating-Point
    - float
    - double
  - Character — char
  - Boolean — boolean
- Non-primitive Data Types
  - String
  - Array
  - List
  - Set
  - Stack
  - Vector
  - Dictionary
  - All user-defined classes
  - etc.,

Data types

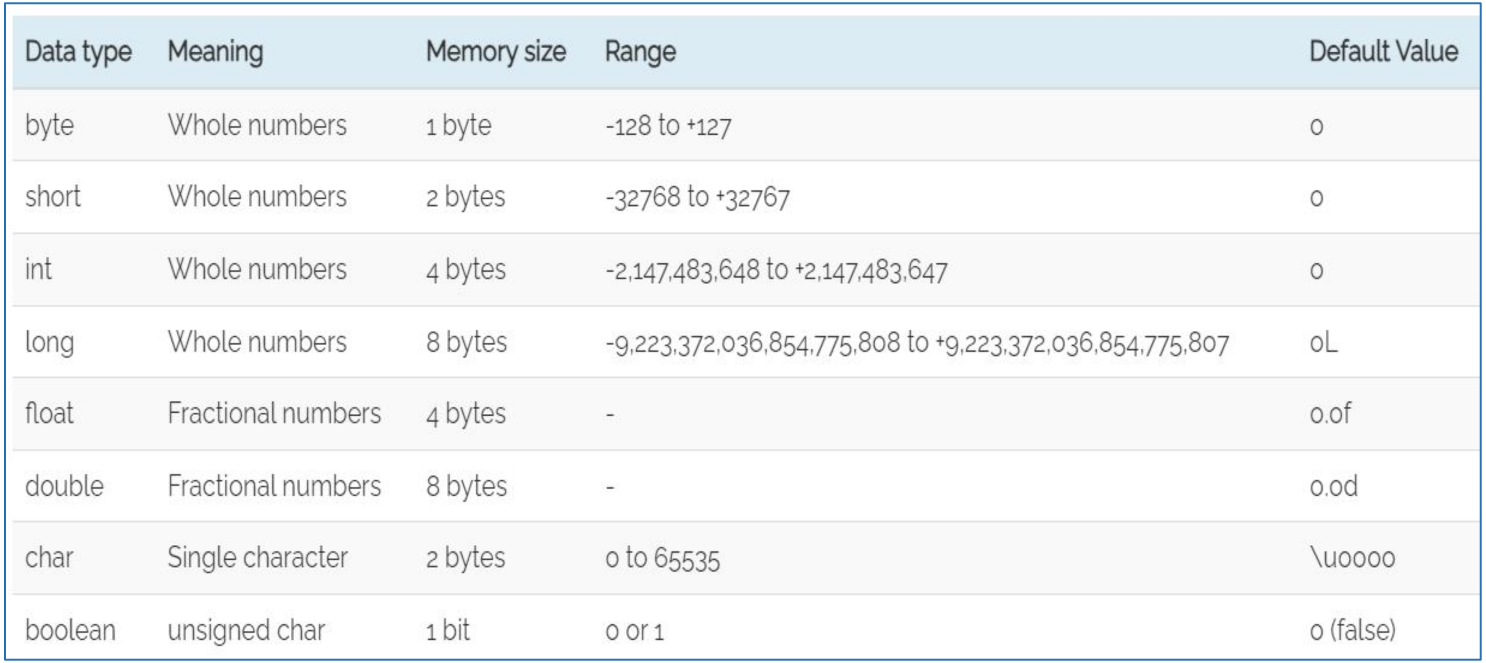
The primitive data types are built-in data types and they specify the type of value stored in a variable and the memory size.

The primitive data types do not have any additional methods.

In Java, primitive data types include

1. Byte
2. Short
3. Int
4. Long
5. Float
6. Double
7. Char
8. boolean

Data types

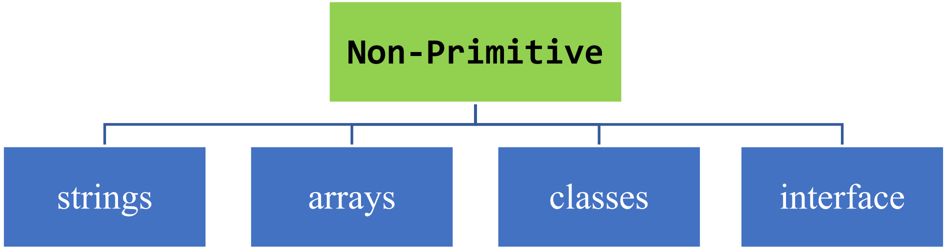


Data type	Meaning	Memory size	Range	Default Value
byte	Whole numbers	1 byte	-128 to +127	0
short	Whole numbers	2 bytes	-32768 to +32767	0
int	Whole numbers	4 bytes	-2,147,483,648 to +2,147,483,647	0
long	Whole numbers	8 bytes	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	0L
float	Fractional numbers	4 bytes	-	0.0f
double	Fractional numbers	8 bytes	-	0.0d
char	Single character	2 bytes	0 to 65535	\u0000
boolean	unsigned char	1 bit	0 or 1	0 (false)

Data types

Non-Primitive Data Type

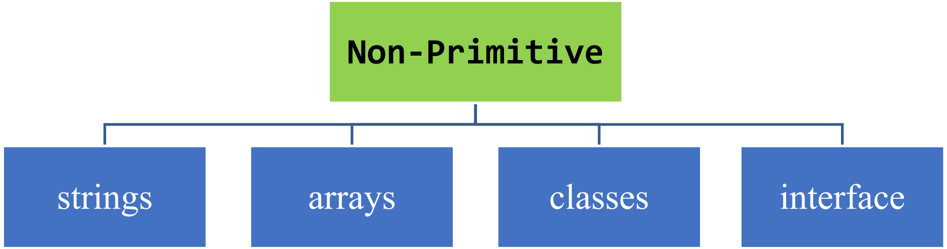
- Strings: String is a sequence of characters. But in Java, a string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.
- Arrays: Arrays in Java are homogeneous data structures implemented in Java as objects. Arrays store one or more values of a specific data type and provide indexed access to store the same. A specific element in an array is accessed by its index.



Data types

- Classes: A class in Java is a blueprint which includes all your data. A class contains fields(variables) and methods to describe the behavior of an object.

- Interface: Like a class, an interface can have methods and variables, but the methods declared in interface are by default abstract (only method signature, no body).



Input Handling

- “Input" refers to the data or information provided to a Java program during its execution.
- This data can be obtained from various sources, such as the user via the keyboard, external files, network connections, and more.
- Input allows programs to interact with users and process dynamic data.

Import the necessary classes

Import the Scanner class:

Import the java.util.Scanner class to enable input reading.

Create a Scanner object:

Initialize a Scanner object with System.in as the argument to read input from the standard input (usually the keyboard).

Prompt the user (optional):

Display a message to guide the user on what input is expected (optional).

Import the necessary classes

Read input:

Use various methods of the Scanner class like next(), nextLine(), nextInt(), nextDouble(), or nextBoolean() to read the input data.

Process the input:

Process the input data as per your program's logic or perform calculations.

Close the Scanner (optional):

Close the Scanner object to release resources (optional, but good practice).

Import the necessary classes

To begin, you need to import the "java.util.Scanner" class, which will allow you to read input from the user. do this at the top of your Java file:

```
import java.util.Scanner;
```

Create scanner object

create a "Scanner" object to read input from various sources, like the standard input stream (usually the keyboard).

initializing a "Scanner" object with "System.in" as the argument.

This associates the "Scanner" with the standard input stream:

```
Scanner scanner = new Scanner(System.in);
```

Import the necessary classes

Prompt the user (optional)

want to display a message to the user, prompting them to enter the input. This step is optional but can be helpful for providing context and guiding the user:

```
System.out.print("Enter your name: ");
```

Read input from user

Read input from the user

To read input from the user, you use the various methods provided by the "Scanner" class. Some commonly used methods include:

"next()": Reads a single word (a sequence of characters separated by whitespace).

"nextLine()": Reads a whole line of text (including spaces).

"nextInt()": Reads the next integer from the input.

"nextDouble()": Reads the next double from the input.

"nextBoolean()": Reads the next boolean value ("true" or "false") from the input.

```
String name = scanner.nextLine();
```

Read input from user

example of reading a line of text (name) from the user:

```
String name = scanner.nextLine();
```

Process the input

Once you have read the input, you can process it as required by your program's logic.

For example, you might want to display the user's name back to them or perform some calculations based on the entered values.

```
System.out.println("Hello, " + name + "!");
```

Close the scanner

It's good practice to close the "Scanner" object when you no longer need it to free up resources. This step is optional, but it's good to include it, especially in larger programs.

```
scanner.close();
```

Example

```
 // Import the Scanner class from the java.util package, which allows reading input from various sources.
import java.util.Scanner;
public class InputExample {
        public static void main(String[] args) {
        // "scanner" to read input from the standard input (keyboard).
        Scanner scanner = new Scanner(System.in);
        // Print a prompt message asking the user to enter their name.
        System.out.print("Enter your name: ");
        // Read the next line of text entered by the user and store it in the variable "name".
        String name = scanner.nextLine();
        // Print a greeting message along with the name entered by the user.
        System.out.println("Hello, " + name + "!");
        // Close the Scanner object to release system resources associated with it (optional but recommended).
        scanner.close();
    }
}
```

Output

"output" refers to the data or information that a Java program produces and displays to the user or writes to an external destination, such as the console, files, network connections, etc.

The process of producing output in Java involves using the “System.out” stream, which is connected to the standard output device, typically the console.

Using "System.out" for Output

In Java, the "System" class provides a static member called "out", which represents the standard output stream. This stream is commonly used to display output to the console. The "out" stream is an instance of the "PrintStream" class, which provides various methods to output data to the console.

.

Printing Text to the Console

To display output to the console, you use the "print()" and "println()" methods of the "PrintStream" class. The "print()" method displays text without moving to the next line, while the "println()" method displays text and moves to the next line after printing.

Printing Variables and Concatenation

You can include variables in the output message using concatenation. The `+` operator is used to concatenate strings and variables together to form a single output.

Formatting Output (Optional)

Java also provides the `printf()` method (inspired by C's `printf`) to format the output. This method allows you to specify placeholders for variables and control the formatting of numbers, text, and other data.

```java
public class OutputExample {
    public static void main(String[] args) {
        // Step 1: Using System.out for Output
        // Step 2: Printing Text to the Console
        System.out.print("This is a "); // Does not move to the next line after printing
        System.out.println("Java program."); // Moves to the next line after printing

        // Step 3: Printing Variables and Concatenation
        String language = "Java";
        int version = 17;
        System.out.println("We are using " + language + " version " + version + ".");

        // Step 4: Formatting Output (Optional)
        double pi = 3.141592653589793;
        System.out.printf("The value of pi is approximately %.2f.%n", pi);
    }
}
```

Explanation

1. "System.out.print("This is a ");": The "print()" method prints the specified text without moving to the next line.

2. "System.out.println("Java program.");": The "println()" method prints the specified text and moves to the next line after printing.

3. "String language = "Java";": Declares a string variable named "language" with the value "Java".

4. "int version = 17;": Declares an integer variable named "version" with the value 17.

5. "System.out.println("We are using " + language + " version " + version + ".");": The "+" operator concatenates the strings and variables to form a single output message.

Explanation

6. "double pi = 3.141592653589793;": Declares a double variable named "pi" with the value of pi (approximately).

7. "System.out.printf("The value of pi is approximately %.2f.%n", pi);": The "printf()" method formats the output to display the value of "pi" with two decimal places. The "%f" is a format specifier for floating-point numbers, and "%.2f" indicates that the value should be displayed with two decimal places. The "%n" is a platform-independent newline character.

Operators

Arithmetic Operators:

+ (addition)

- (subtraction)

* (multiplication)

/ (division)

% (modulo or remainder)

Operators

1. Addition (+):

The addition operator is used to add two numeric values together.

Example:

```
int num1 = 5;

int num2 = 3;

int sum = num1 + num2; // sum will be 8
```

Operators

2. Subtraction (-):

The subtraction operator is used to subtract one numeric value from another.

Example:

```
int num1 = 10;

int num2 = 4;

int difference = num1 - num2; // difference will be 6
```

Operators

3. Multiplication (*):

The multiplication operator is used to multiply two numeric values.

Example:

```
int num1 = 6;

int num2 = 7;

int product = num1 * num2; // product will be 42
```

Operators

4. Division (/):

The division operator is used to divide one numeric value by another. Note that if both operands are integers, the result will also be an integer, and any fractional part will be truncated.

Example:

```
int num1 = 15;

int num2 = 4;

int quotient = num1 / num2; // quotient will be 3 (integer division)
```

Operators

If you want to get the exact result with decimal points, you can use floating-point data types like `float` or `double`.

Example (using `double` for decimal division):

```
double num1 = 15.0;

double num2 = 4.0;

double quotient = num1 / num2; // quotient will be 3.75
```

Operators

5. Modulo/Remainder (%):

The modulo operator returns the remainder of the division operation between two numeric values. It is often used to determine if a number is even or odd (by checking if the remainder is 0 or 1, respectively).

Example:

```
int num1 = 17;

int num2 = 5;

int remainder = num1 % num2; // remainder will be 2
```

In this example, 17 divided by 5 equals 3 with a remainder of 2.

Operators

Relational Operators:

Relational operators in Java are used to compare two values or expressions and evaluate their relationship. They return a boolean result, either true or false, based on the comparison.

1. == (equal to)
2. != (not equal to)
3. > (greater than)
4. < (less than)
5. >= (greater than or equal to)
6. <= (less than or equal to)

Operators

Relational Operators:

Relational operators in Java are used to compare two values or expressions and evaluate their relationship. They return a boolean result, either true or false, based on the comparison.

1. == (equal to)
2. != (not equal to)
3. > (greater than)
4. < (less than)
5. >= (greater than or equal to)
6. <= (less than or equal to)

Operators

1. == (equal to): This operator checks if two operands are equal or not.

Example:

```
int a = 5;

int b = 5;

boolean result = (a == b); // true, as both 'a' and 'b' have the same value (5).
```

Operators

2. != (not equal to): This operator checks if two operands are not equal.

Example:

int x = 10;

int y = 5;

boolean result = (x != y); // true, as 'x' and 'y' have different values (10 and 5).

Operators

3. > (greater than): This operator checks if the left operand is greater than the right operand.

Example:

```
int p = 7;

int q = 3;

boolean result = (p > q); // true, as 'p' is greater than 'q'.
```

Operators

4. < (less than): This operator checks if the left operand is less than the right operand.

Example:

```
int m = 4;

int n = 8;

boolean result = (m < n); // true, as 'm' is less than 'n'.
```

Operators

5. >= (greater than or equal to): This operator checks if the left operand is greater than or equal to the right operand.

Example:

int num1 = 6;

int num2 = 6;

boolean result = (num1 >= num2); // true, as 'num1' is equal to 'num2' (6) and therefore greater than or equal to 'num2'.

Operators

6. <= (less than or equal to): This operator checks if the left operand is less than or equal to the right operand.

Example:

int value1 = 4;

int value2 = 9;

boolean result = (value1 <= value2); // true, as 'value1' is less than 'value2'.

Operators

Logical Operators:

&& (logical AND)

|| (logical OR)

! (logical NOT)

Operators

1. &&(logical AND):

- The logical AND operator returns “true” if and only if both operands are "true". If any of the operands is "false", the result will be "false".

- It is also known as the short-circuit AND because if the left operand evaluates to "false", the right operand will not be evaluated since the result will already be "false".

Operators

2. ||(logical OR):

- The logical OR operator returns "trueif at least one of the operands is "true". It returns "falseonly when both operands are "false".

- Like "&&", it is also short-circuit OR. If the left operand evaluates to "true", the right operand will not be evaluated since the result will already be "true".

3. !(logical NOT):

- The logical NOT operator reverses the boolean value of its operand. If the operand is "true", "!will make it "false", and if the operand is "false", "!will make it "true".

Operators

```
public class LogicalOperators {
    public static void main(String[] args) {
        int age = 25;
        boolean isStudent = true;
        // Using && (logical AND)
        boolean isAdultStudent = age >= 18 && isStudent;
        // The result will be true if age is greater than or equal to 18 AND isStudent is true.
        // Using || (logical OR)
        boolean isAdultOrStudent = age >= 18 || isStudent;
        // The result will be true if age is greater than or equal to 18 OR isStudent is true.
        // Using ! (logical NOT)
        boolean isNotStudent = !isStudent;
        // The result will be false since we negate the value of isStudent
        // Printing the results
        System.out.println("Is the person an adult student? " + isAdultStudent);
        System.out.println("Is the person either an adult or a student? " + isAdultOrStudent);
        System.out.println("Is the person not a student? " + isNotStudent);
    }
}
```

Operators

Operator is a symbol that is used to perform operations. For example: +, -, *, / etc. some common types of operators in Java:

- Arithmetic Operators:

+ (addition)

- (subtraction)

* (multiplication)

/ (division)

% (modulo or remainder)

- Relational Operators:

== (equal to)

!= (not equal to)

> (greater than)

< (less than)

>= (greater than or equal to)

<= (less than or equal to)

Operators

- Logical Operators:

&& (logical AND)

|| (logical OR)

! (logical NOT)

- Assignment Operators:

= (simple assignment)

+= (add and assign)

-= (subtract and assign)

*= (multiply and assign)

/= (divide and assign)

%= (modulo and assign)

- Increment/Decrement Operators:

++ (increment)

-- (decrement)

Operators

- Bitwise Operators:

& (bitwise AND)

| (bitwise OR)

^ (bitwise XOR)

~ (bitwise NOT)

<< (left shift)

>> (right shift)

>>> (unsigned right shift)

- Conditional (Ternary) Operator:

? : (conditional operator, also known as the ternary operator)

Operators

Assignment Operators:

Assignment operators are used to assign values to variables.

1. = (simple assignment)
2. += (add and assign)
3. -= (subtract and assign)
4. *= (multiply and assign)
5. /= (divide and assign)
6. %= (modulo and assign)

Operators

1. "=" (Simple Assignment):

The "=" operator is the basic assignment operator in . It assigns the value of the right-hand operand to the left-hand operand.

Example:

int x = 10; // Assign the value 10 to the variable 'x'

2. "+=" (Add and Assign):

The "+=" operator adds the value of the right-hand operand to the left-hand operand and then assigns the result to the left-hand operand.

Example:

int a = 5;

a += 3; // Equivalent to: a = a + 3; Now 'a' becomes 8

Operators

3. "-=" (Subtract and Assign):

The "-=" operator subtracts the value of the right-hand operand from the left-hand operand and then assigns the result to the left-hand operand.

Example:

int b = 10;

b -= 4; // Equivalent to: b = b - 4; Now 'b' becomes 6

4. "*=" (Multiply and Assign):

The "*= operator multiplies the value of the left-hand operand by the right-hand operand and then assigns the result to the left-hand operand.

Example:

int c = 3;

c *= 2; // Equivalent to: c = c * 2; Now 'c' becomes 6

5. "/=" (Divide and Assign):

The "/=" operator divides the value of the left-hand operand by the right-hand operand and then assigns the result to the left-hand operand.

Example:

```
int d = 15;

d /= 3; // Equivalent to: d = d / 3; Now 'd' becomes 5
```

6. "%=" (Modulo and Assign):

The "%=" operator calculates the modulo of the left-hand operand with the right-hand operand and then assigns the result to the left-hand operand.

Example:

```
int e = 7;

e %= 4; // Equivalent to: e = e % 4; Now 'e' becomes 3
```

Operators

Increment/Decrement Operators:

++ (increment)

-- (decrement)

Operators

1. `++` (increment): The increment operator is used to increase the value of a variable by 1.

Example:

"int count = 5;

count++; // This is equivalent to count = count + 1;

System.out.println(count); // Output: 6"

In the example above, the value of the `count` variable starts at 5, and after applying the increment operator (`count++`), its value becomes 6.

Operators

2. `--` (decrement): The decrement operator is used to decrease the value of a variable by 1.

Example:

```
"int quantity = 10;

quantity--; // This is equivalent to quantity = quantity - 1;

System.out.println(quantity); // Output: 9"
```

Operators

Bitwise Operators:

1. & (bitwise AND)
2. | (bitwise OR)
3. ^ (bitwise XOR)
4. ~ (bitwise NOT)
5. << (left shift)
6. >> (right shift)
7. >>> (unsigned right shift)

Operators

1. "&" (bitwise AND): Performs a bitwise AND operation between the binary representations of two integers. Each bit of the result is set to 1 only if the corresponding bits of both operands are 1.

Example:

```
int num1 = 12;   // Binary: 1100
int num2 = 10;   // Binary: 1010
int result = num1 & num2;  // Binary result: 1000 (8 in decimal)
```

Operators

2. "|" (bitwise OR): Performs a bitwise OR operation between the binary representations of two integers. Each bit of the result is set to 1 if at least one of the corresponding bits in either operand is 1.

Example:

```
int num3 = 12;   // Binary: 1100
int num4 = 10;   // Binary: 1010
int result = num3 | num4;  // Binary result: 1110 (14 in decimal)
```

Operators

2. "|" (bitwise OR): Performs a bitwise OR operation between the binary representations of two integers. Each bit of the result is set to 1 if at least one of the corresponding bits in either operand is 1.

Example:

```
int num3 = 12;   // Binary: 1100
int num4 = 10;   // Binary: 1010
int result = num3 | num4;  // Binary result: 1110 (14 in decimal)
```

Operators

3. "^" (bitwise XOR): Performs a bitwise XOR (exclusive OR) operation between the binary representations of two integers. Each bit of the result is set to 1 if the corresponding bits of the operands are different (one 0 and one 1).

Example:

```
int num5 = 12;   // Binary: 1100

int num6 = 10;   // Binary: 1010

int result = num5 ^ num6;  // Binary result: 0110 (6 in decimal)
```

Operators

4. "~" (bitwise NOT): Performs a bitwise NOT operation, which flips the bits of an integer.

Each 0 bit in the original number becomes 1, and each 1 bit becomes 0.

Example:

```
int num7 = 12;   // Binary: 1100

int result = ~num7;  // Binary result: 0011 (3 in decimal)
```

Operators

5. "<<" (left shift): Shifts the bits of an integer to the left by the specified number of positions. It effectively multiplies the number by 2 raised to the power of the shift count.

Example:

```
int num8 = 5;   // Binary: 00000101

int result = num8 << 2;  // Binary result: 00010100 (20 in decimal)
```

Operators

6. ">>" (right shift): Shifts the bits of an integer to the right by the specified number of positions. The leftmost bit is filled with the sign bit (in case of a signed data type) or with zeros (in case of an unsigned data type).

Example:

```
int num9 = -8;   // Binary: 11111000
int result = num9 >> 2;  // Binary result: 11111110 (-2 in decimal)
```

Operators

7. ">>>" (unsigned right shift): Similar to ">>", but the leftmost bits are always filled with zeros, regardless of the sign of the number.

Example:

```
int num10 = -8;   // Binary: 11111000
int result = num10 >>> 2;  // Binary result: 00111110 (62 in decimal)
```

Operators

Conditional (Ternary) Operator:

The conditional operator, also known as the ternary operator, is a compact way to express a simple if-else statement in Java. It has the following syntax:

condition ? expression1 : expression2

The condition is evaluated first, and if it is true, then expression1 is returned; otherwise, expression2 is returned. The conditional operator is useful for assigning a value based on a condition in a concise manner.

Operators

Example:

```
int x = 10;
int y = 5;
int result = (x > y) ? x : y;
// The above line is equivalent to the following if-else statement:
// int result;
// if (x > y) {
//     result = x;
// } else {
//     result = y;
// }
```

In this example, the result variable will be assigned the value of x (which is 10) because the condition `x > y` is true.

Operators

Example:

```
int x = 10;
int y = 5;
int result = (x > y) ? x : y;
// The above line is equivalent to the following if-else statement:
// int result;
// if (x > y) {
//     result = x;
// } else {
//     result = y;
// }
```

In this example, the result variable will be assigned the value of x (which is 10) because the condition `x > y` is true.

What is decision-making?

Decision-making statements are used to control the flow of a program based on certain conditions. These statements allow you to make decisions and execute different blocks of code depending on whether a given condition is true or false.

1. if statement: The "if" statement is the fundamental decision-making statement. It evaluates a boolean expression inside the parentheses and executes the block of code within the curly braces if the condition is true.

```
if (condition) {
   // Code to be executed if the condition is true
}
```

Example

```
public class Main {
    public static void main(String[] args) {
        int number = 10;
        if (number > 5) {
            System.out.println("The number is greater than 5.");
        }
    }
}
```

2.if-else statement: The "if-else" statement provides an alternative block of code to be executed when the condition is false.

```
if (condition) {
   // Code to be executed if the condition is true
} else {
   // Code to be executed if the condition is false
}
```

```
public class Main {
    public static void main(String[] args) {
        int number = 3;
        if (number % 2 == 0) {
            System.out.println("The number is even.");
        } else {
            System.out.println("The number is odd.");
        }
    }
}
```

3. if-else if-else statement: The "if-else if-else" statement allows you to chain multiple conditions together and execute different blocks of code based on the first condition that evaluates to true.

```
if (condition1) {
    // Code to be executed if condition1 is true
} else if (condition2) {
    // Code to be executed if condition 2 is true
} else {
    // Code to be executed if all previous conditions are false
}
```

```
public class Main {
    public static void main(String[] args) {
        int score = 85;
        if (score >= 90) {
            System.out.println("A");
        } else if (score >= 80) {
            System.out.println("B");
        } else if (score >= 70) {
            System.out.println("C");
        } else {
            System.out.println("D");
        }}}
```

4. switch statement: The "switch" statement provides an alternative way to handle multiple conditions based on the value of an expression. It allows you to choose a specific block of code to execute based on different possible values.

```
switch (expression) {
   case value1:
      // Code to be executed if the expression equals value1
      break;
   case value2:
      // Code to be executed if the expression equals value2
      break;
   // More cases can be added here
   default:
      // Code to be executed if none of the cases match the expression
}
```

```
public class Main {
    public static void main(String[] args)
{
        int dayOfWeek = 2;
        String dayName;
        switch (dayOfWeek) {
            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);
    }
}
```

decision making statements

Ternary operator (?:):

It is a shorthand way to write simple if-else statements.

```
int num = 10;
String result = (num > 0) ? "Positive" : "Non-positive";
System.out.println(result);
```

decision making statements-Jump statements

There are three jump statements:

- Break
- Continue
- Return

These statements are used to control the flow of a program and are typically used in loops and conditional blocks.

decision making statements-Jump statements

1. break: The `break` statement is used to exit a loop prematurely, even if the loop condition is not met. When the `break` statement is encountered, the control flow exits the loop, and the program continues with the statement after the loop.

Example

```
public class Main {
    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++) {
            if (i == 5) {
                break; // Exit the loop when i becomes 5
            }
            System.out.println("Value: " + i);
        }
    }
}
```

decision making statements-Jump statements

2. continue: The `continue` statement is used to skip the rest of the current iteration and continue with the next iteration of a loop. When the `continue` statement is encountered, the control flow jumps back to the loop's beginning to evaluate the loop condition again.

Example

```
public class Main {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            if (i == 3) {
                        continue; // Skip iteration when i is 3 and continue with the next
iteration
            }
            System.out.println("Value: " + i);
        }
    }
}
```

decision making statements-Jump statements

3.return: The `return` statement is used inside a method to terminate the method's execution and optionally return a value to the caller. When a `return` statement is encountered, the method's execution stops, and the control flow returns to the calling method.

Example

```
public class Main {
    public static void main(String[] args) {
        int num1 = 10;
        int num2 = 20;
        int result = add(num1, num2);
        System.out.println("Sum: " + result);
    }
    public static int add(int a, int b) {
        int sum = a + b;
        return sum; // Return the sum to the caller
    }
}
```

What is an Algorithm Complexity?

The complexity of an algorithm in Java, or any programming language, refers to the evaluation of its efficiency in terms of time and space requirements as a function of the input size. As mentioned earlier, algorithm complexity is typically categorized into two types:

a. Time Complexity: It measures the amount of time an algorithm takes to run as a function of the input size 'n'. Time complexity is denoted using Big O notation, which provides an upper bound on the growth rate of the algorithm's running time. The notation is expressed as O(f(n)), where 'f(n)' represents a function describing the worst-case time required for an algorithm.

b. Space Complexity: It measures the amount of memory space an algorithm uses as a function of the input size 'n'. Space complexity is also denoted using Big O notation, and it represents the upper bound on the additional memory space required by the algorithm during execution.

Analysis of Algorithm

- The goal of analysis of an algorithm is to compare algorithm in running time and also Memory management.
- Running time of an algorithm depends on how long it takes a computer to run the lines of code of the algorithm.

Running time of an algorithm depends on

1.Speed of computer

2.Programming language

3.Compiler and translator

Examples: binary search, linear search

Asymptotic Analysis:

- Expressing the complexity in terms of its relationship to know function. This type of analysis is called asymptotic analysis.
- The main idea of Asymptotic analysis is to have a measure of the efficiency of an algorithm, that doesn’t depends on

1.Machine constants.

2.Doesn’t require an algorithm to be implemented.

3.Time taken by the program to be prepared.

Asymptotic Notation:

The mathematical way of representing the Time complexity.

- The notation we use to describe the asymptotic running time of an algorithm is defined in terms of functions whose domains are the set of natural numbers.

Definition: It is the way to describe the behavior of functions in the limit or without bounds.

- Asymptotic growth: The rate at which the function grows…
- “growth rate” is the complexity of the function or the amount of resources it takes up to compute.
- **Growth rate** **Time +memory**

→

Classification of growth:

1.Growing with the same rate.

2.Growing with the slower rate.

3.Growing with the faster rate.

types:

They are 3 asymptotic notations are mostly used to represent the time complexity of the algorithm.

1.Big oh (O)notation

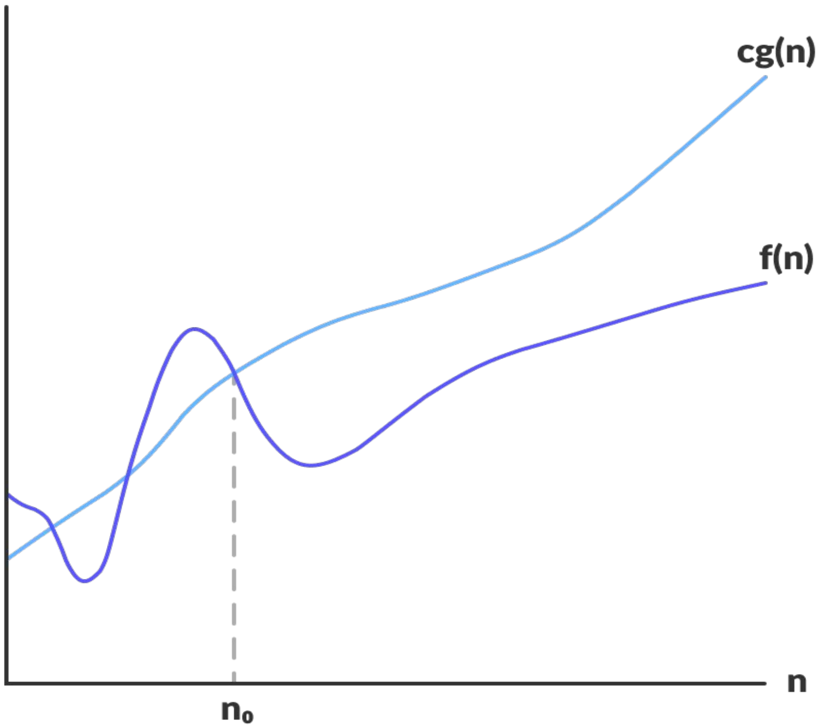
2.Big omega (Ω) notation

3.Theta(Θ) notation

Big-O notation

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.

Big-O notation



f(n) = O(g(n))

Big-O notation

O(g(n)) = { f(n): there exist positive constants c and n0

such that 0 ≤ f(n) ≤ cg(n) for all n ≥ n0 }

The above expression can be described as a function f(n) belongs to the set O(g(n)) if there exists a positive constant c such that it lies between 0 and cg(n), for sufficiently large n.

For any value of n, the running time of an algorithm does not cross the time provided by O(g(n)).

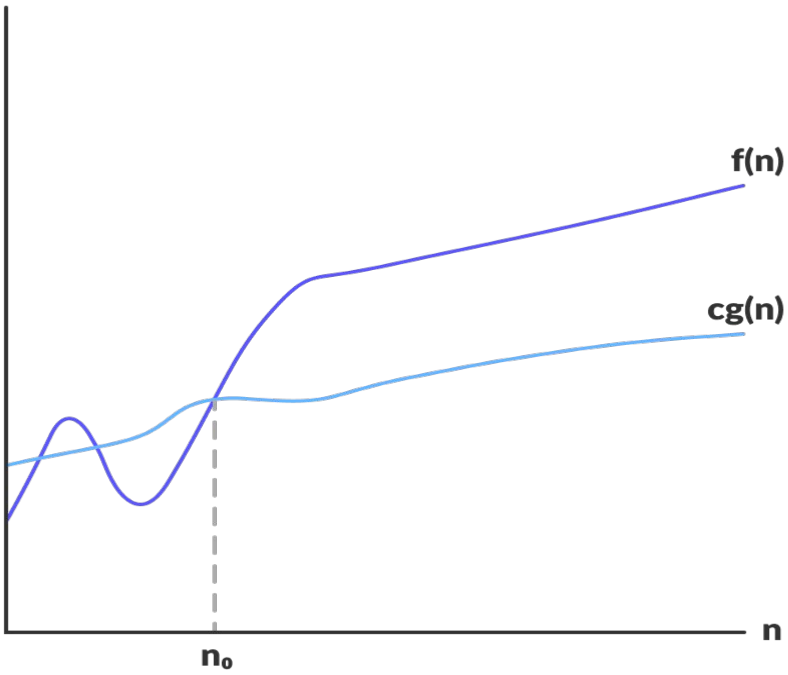
Big-O notation

Since it gives the worst-case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst-case scenario.

Omega(Ω) notation

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.

Omega(Ω) notation



$f(n) = \Omega(g(n))$

Omega(Ω) notation

Ω(g(n)) = { f(n): there exist positive constants c and n0

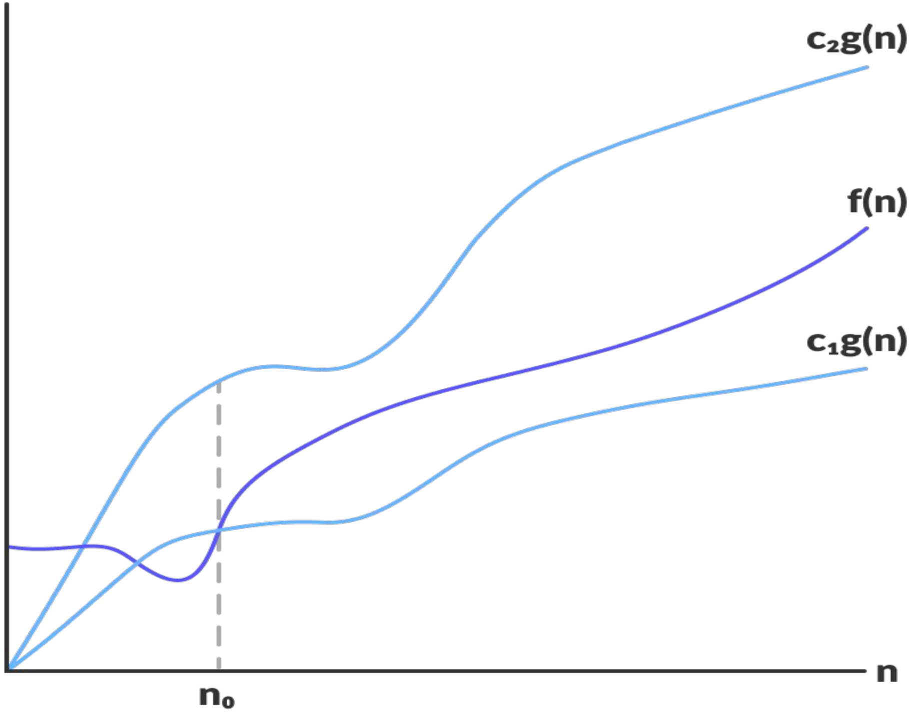
such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n0$ }

The above expression can be described as a function f(n) belongs to the set Ω(g(n)) if there exists a positive constant c such that it lies above cg(n), for sufficiently large n. For any value of n, the minimum time required by the algorithm is given by Omega Ω (g(n)).

Theta Notation (Θ-notation)

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.

Theta Notation (Θ-notation)



$f(n) = \Theta(g(n))$

Theta Notation (Θ-notation)

For a function g(n), Θ(g(n)) is given by the relation:

$\Theta(g(n)) = \{ f(n)$: there exist positive constants c1, c2 and n0

such that $0 \le c1g(n) \le f(n) \le c2g(n)$ for all $n \ge n0 \}$

The above expression can be described as a function f(n) belongs to the set Θ(g(n)) if there exist positive constants c1 and c2 such that it can be sandwiched between c1g(n) and c2g(n), for sufficiently large n.

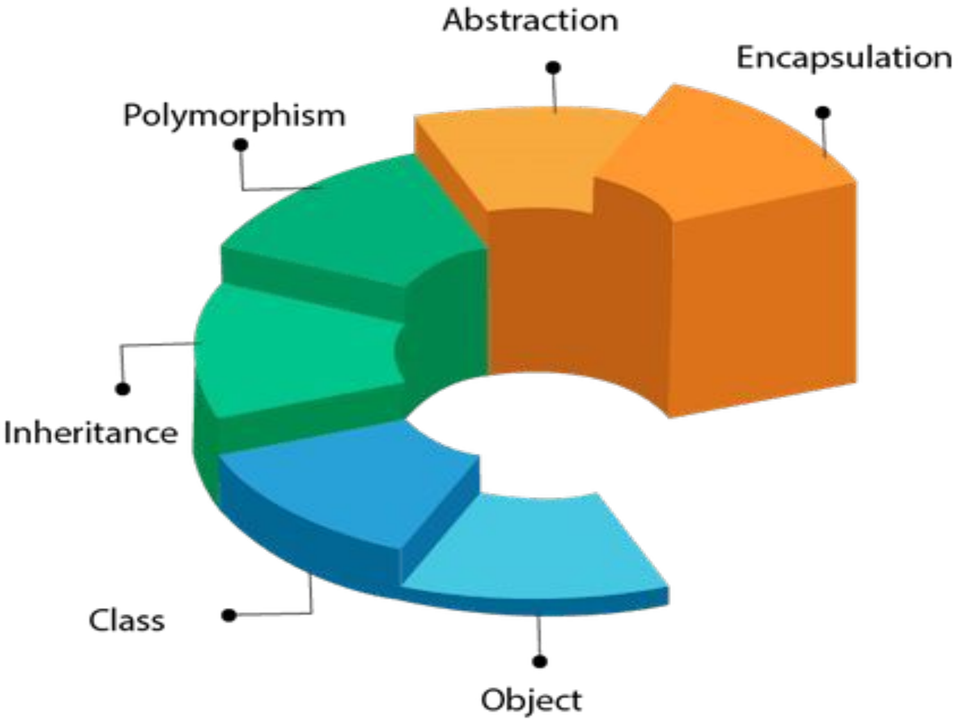
Theta Notation (Θ-notation)

If a function f(n) lies anywhere in between c1g(n) and c2g(n) for all n ≥ n0, then

f(n) is said to be asymptotically tight bound.

OOPS

OOPs (Object-Oriented Programming System)

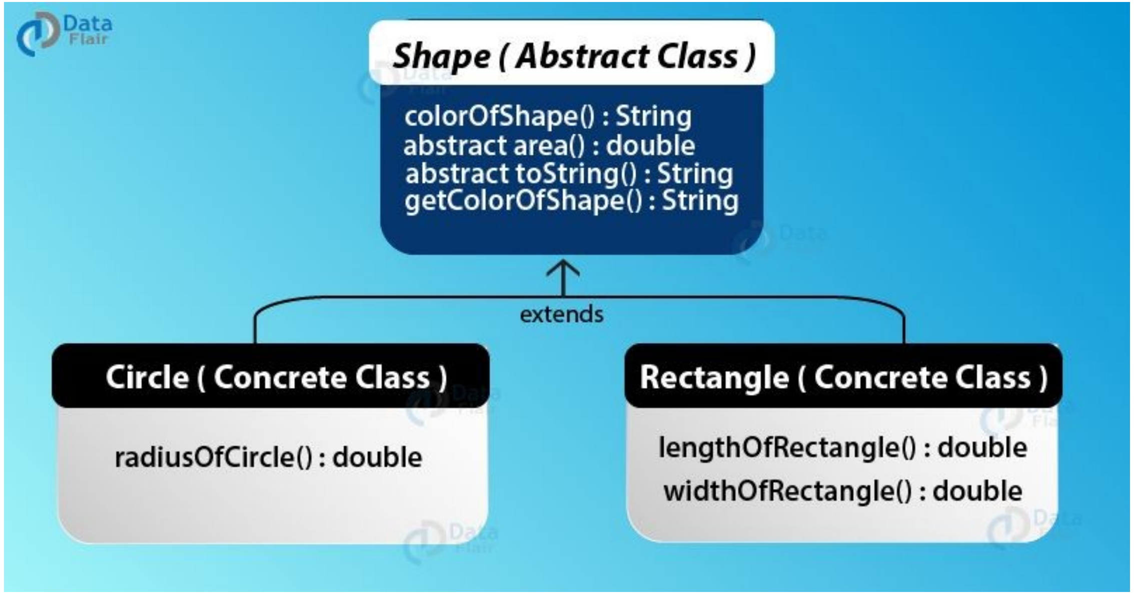


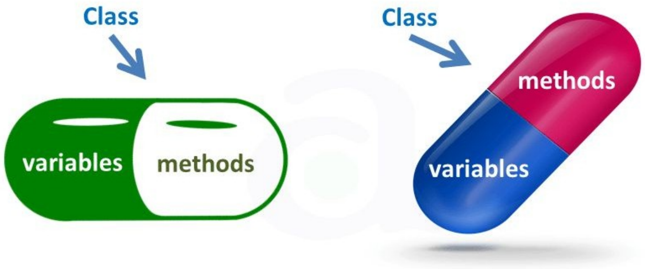
CLASS ABSTRACTION

- Abstraction is the quality of dealing with ideas rather than events. For example, when you consider the case of e-mail, complex details such as what happens as soon as you send an e-mail, the protocol your e-mail server uses are hidden from the user. Therefore, to send an e-mail you just need to type the content, mention the address of the receiver, and click send.
- Likewise in Object-oriented programming, abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it.

ENCAPSULATION

- Encapsulation is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.
- Encapsulation in Java is a mechanism for wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes and can be accessed only through the methods of their current class. Therefore, it is also known as data hiding.
- To achieve encapsulation in Java –
  - Declare the variables of a class as private.
  - Provide public setter and getter methods to modify and view the variables values.

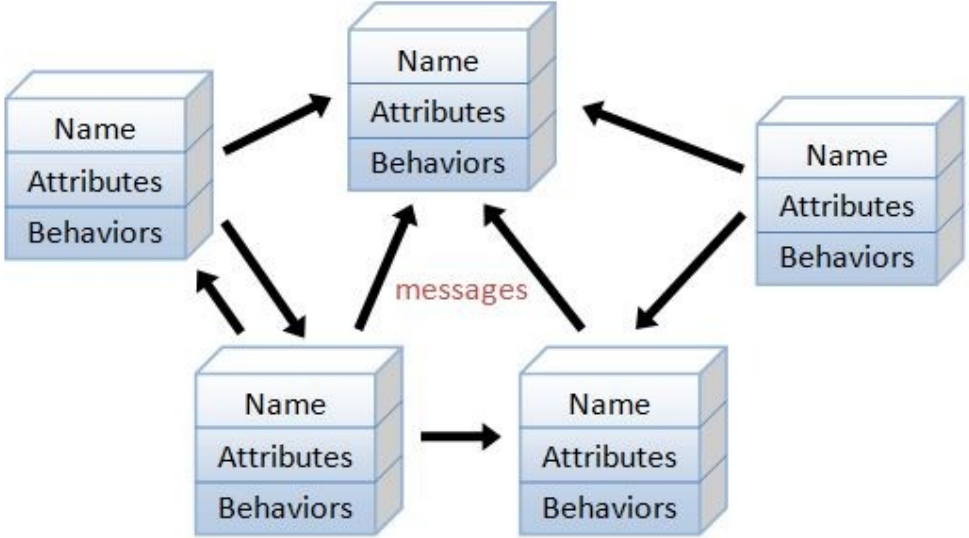

Data Flair
Shape (Abstract Class)
colorOfShape() : String
abstract area() : double
abstract toString() : String
getColorOfShape() : String
extends
Circle (Concrete Class)
radiusOfCircle() : double
Rectangle (Concrete Class)
lengthOfRectangle() : double
widthOfRectangle() : double



Encapsulation

OBJECTS

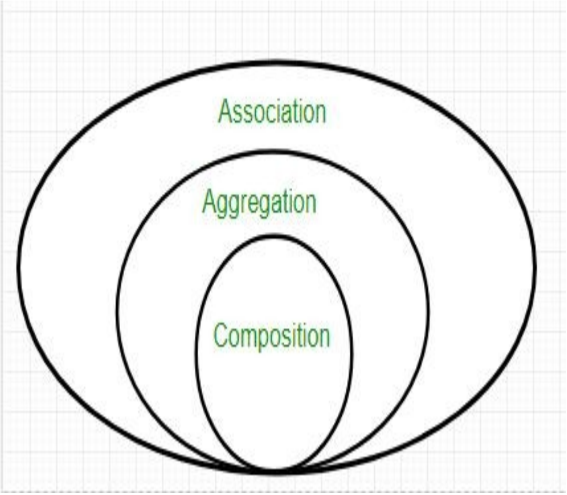
- ❖ The **object oriented Programming** Language is based upon the concept of “**objects**”, which contains data as attributes in methods. Every **object** in **Java** has state and behavior which are represented by instance variables and methods. ... Here method is using instance variable values.



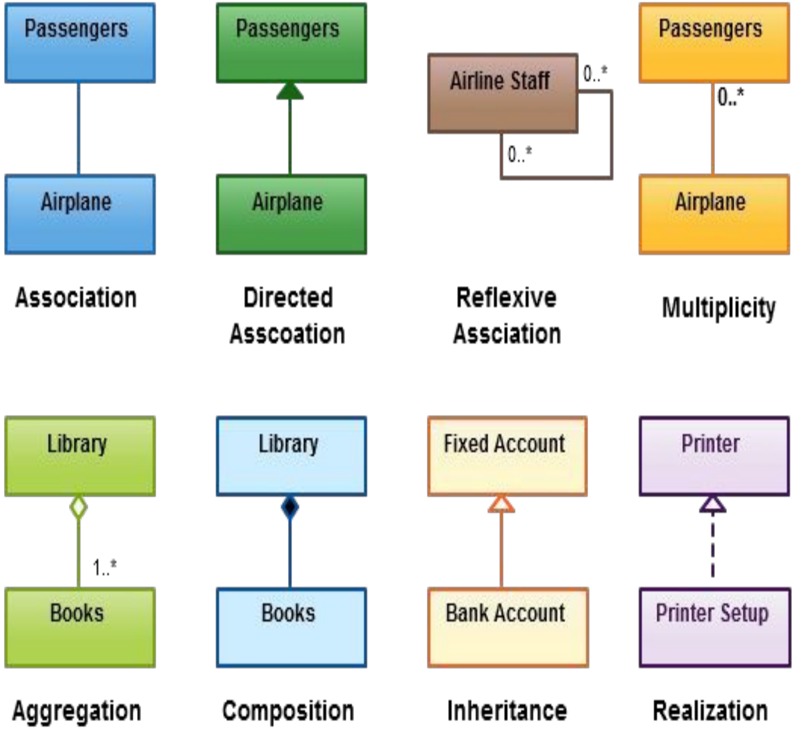
An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages

CLASS RELATIONSHIPS

- Association is relation between two separate classes which establishes through their Objects. Association can be
  one-to-one, one-to-many, many-to-one,
  many-to-many.
- In Object-Oriented programming, an Object communicates to other Object to use functionality and services provided by that object. **Composition** and **Aggregation** are the two forms of association.
- It is a special form of Association where:
  - It represents **Has-A** relationship.



- It is a **unidirectional association** i.e. a one way relationship. For example, a department can have students but vice versa is impossible and thus unidirectional.
- In Aggregation, **both entries can survive individually** which means ending one entity will not affect the other entity



- Composition is a restricted form of Aggregation in which two entities are highly dependent on each other.
  - It represents **part-of** relationship.
  - In composition, both the entities are dependent on each other.
  - When there is a composition between two entities, the composed object **cannot exist** without the other entity.

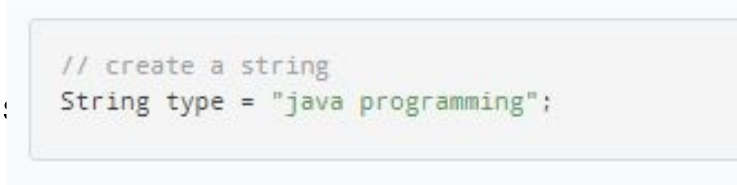
STRING CLASS

- In Java, a string is a sequence of characters. For example, "hello" is a string containing a sequence of

  characters 'h', 'e', 'l', 'l', and 'o'.
- Unlike other programming languages, strings in Java are not primitive types (like int, char, etc).

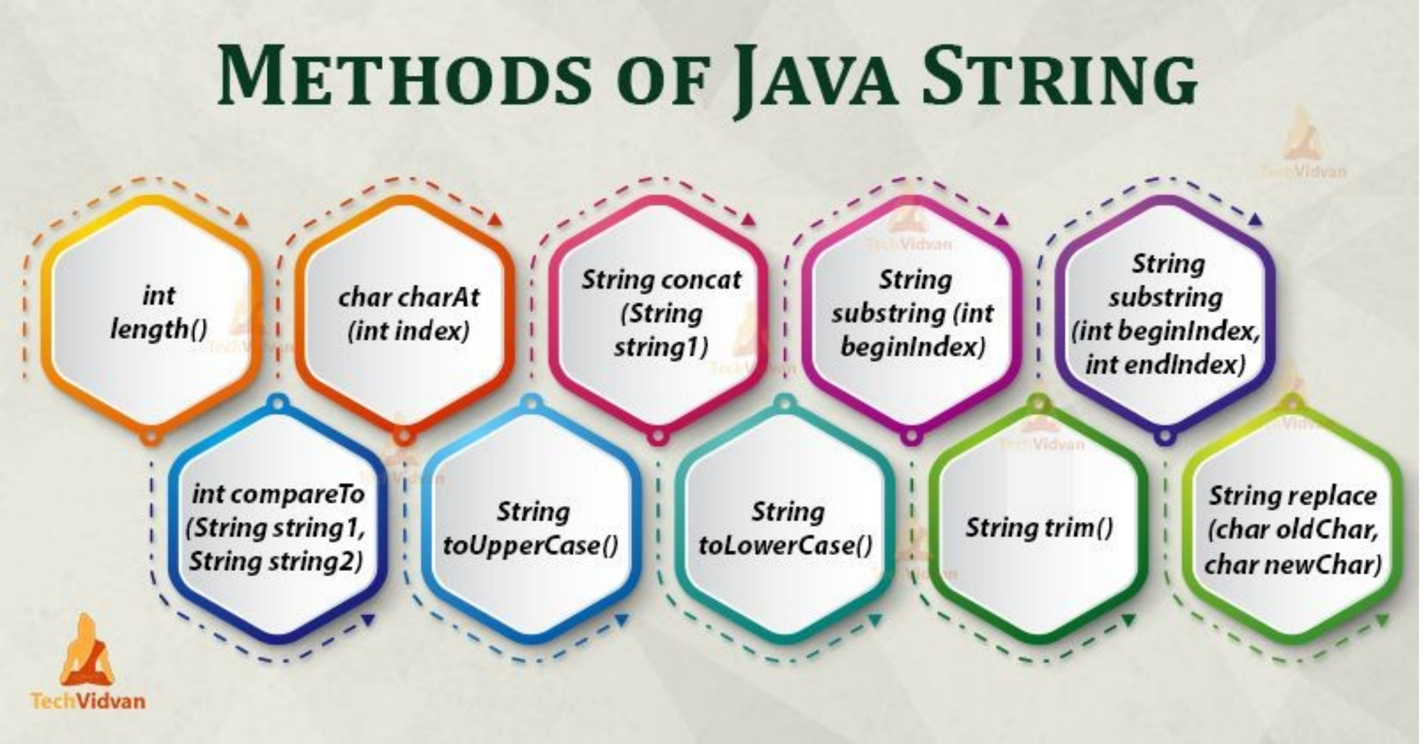
  Instead, all strings are objects of a predefined class named String. For example,
- Here, we have created a string named type. Here, we have initialized the string with "java

  programming". In Java, we use double quotes to represent a string.
- The string is an ins

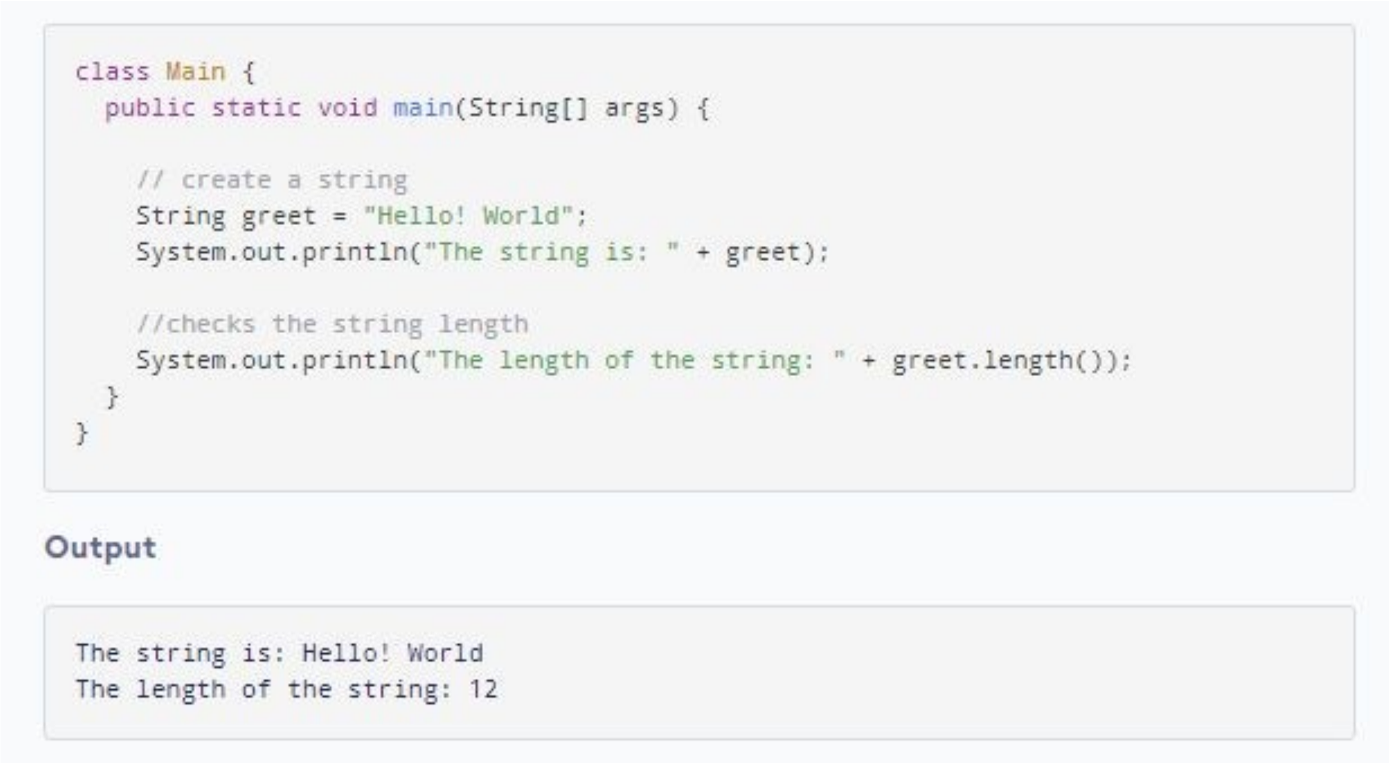


```
// create a string
String type = "java programming";
```

STRING CLASS


METHODS OF JAVA STRING
int length()
char charAt (int index)
String concat (String string1)
String substring (int beginIndex)
String substring (int beginIndex, int endIndex)
int compareTo (String string1, String string2)
String toUpperCase()
String toLowerCase()
String trim()
String replace (char oldChar, char newChar)
TechVidvan

Example 1: Java find string's length

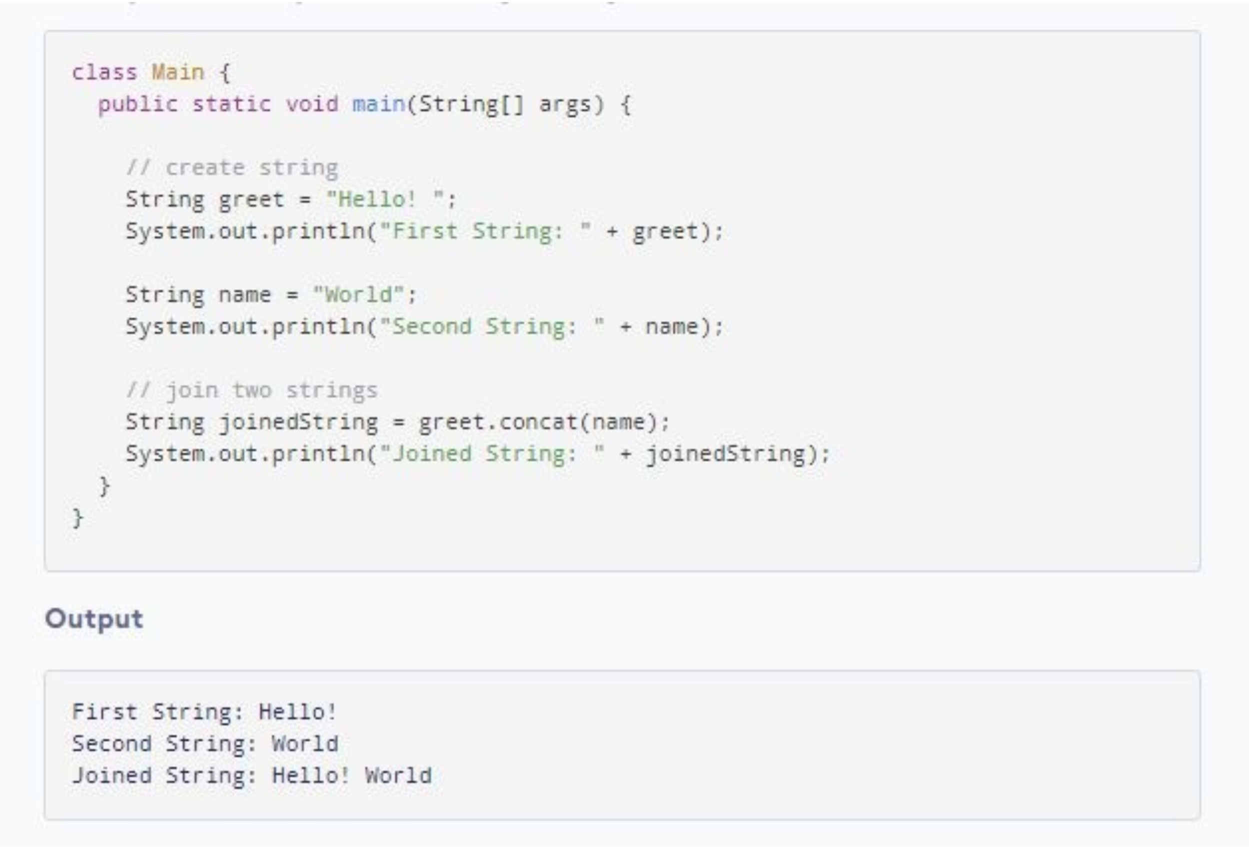


```java
class Main {
  public static void main(String[] args) {

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

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

Output

```
The string is: Hello! World
The length of the string: 12
```

Example 2: Java join two strings using concat()

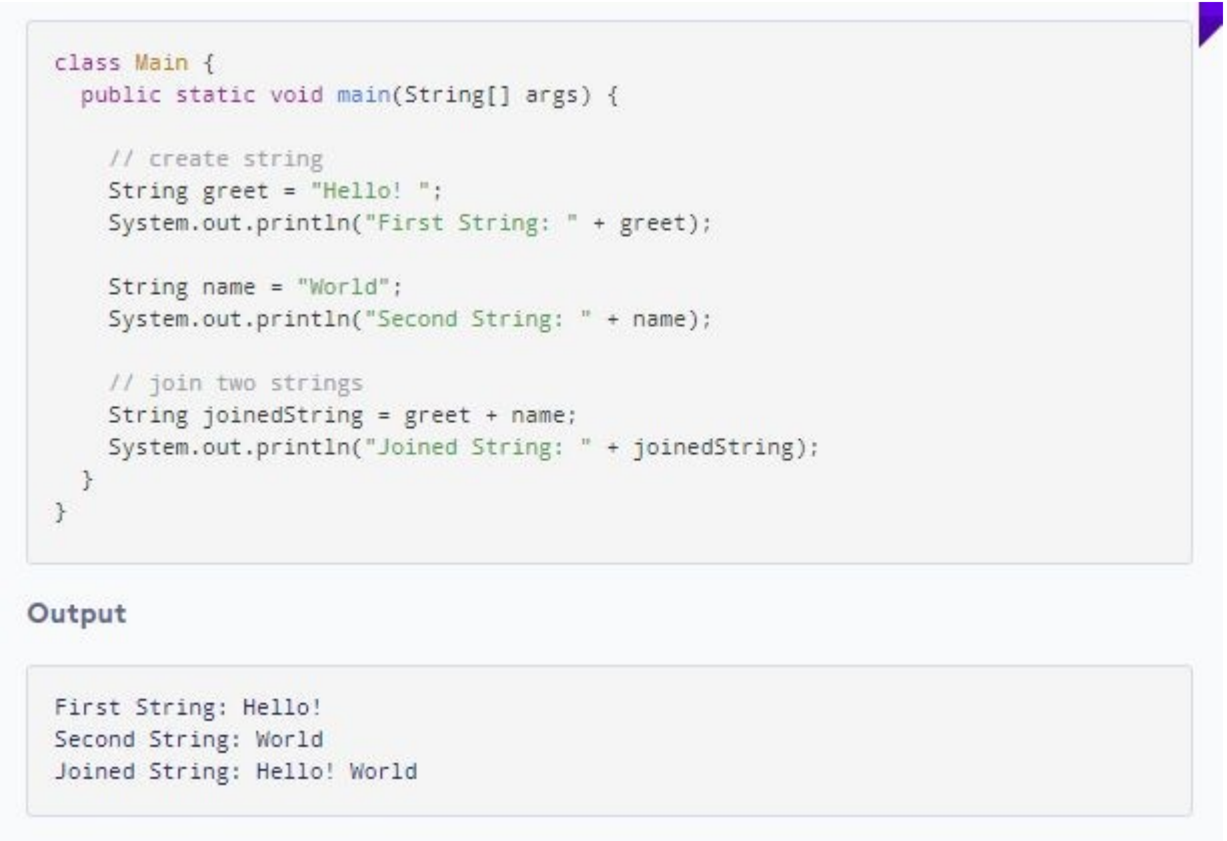


```
class Main {
  public static void main(String[] args) {

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

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

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

Output

```
First String: Hello!
Second String: World
Joined String: Hello! World
```

Example 3: Java join strings using + operator

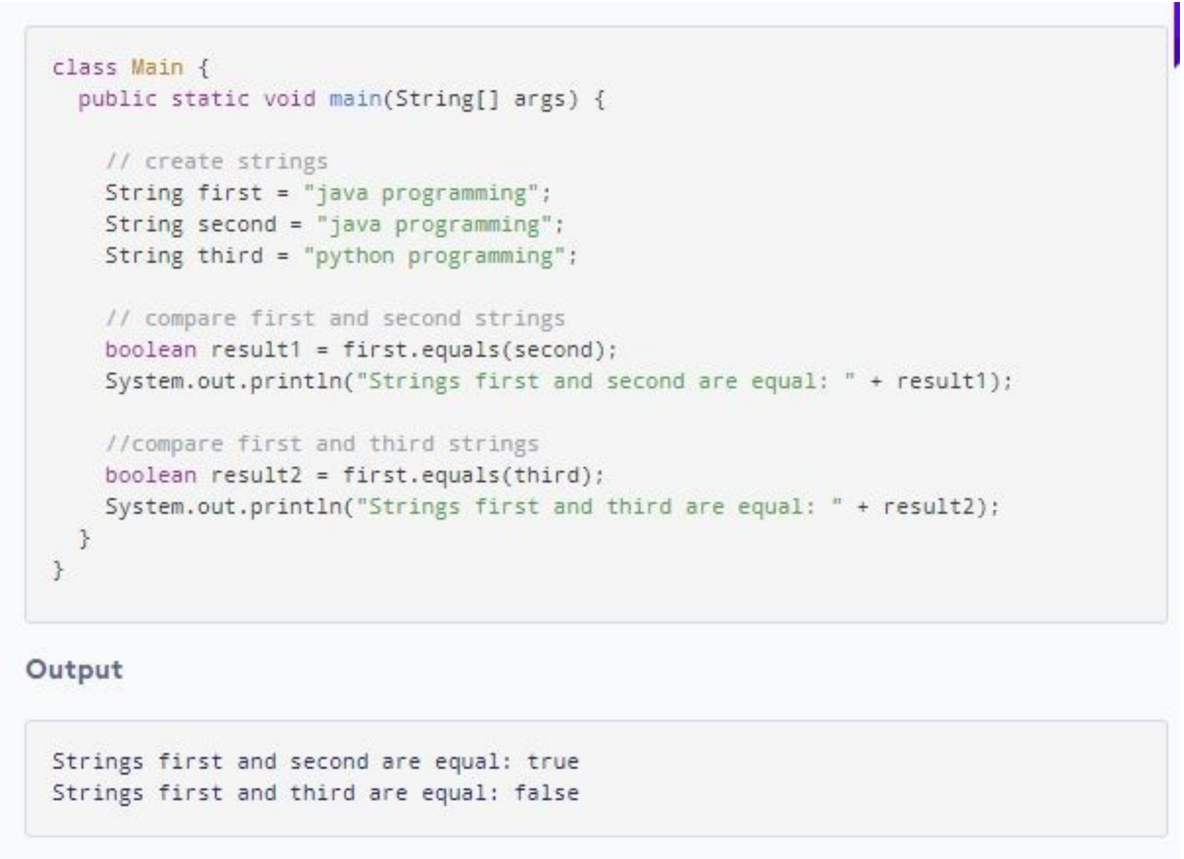


```java
class Main {
  public static void main(String[] args) {

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

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

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

Output

```
First String: Hello!
Second String: World
Joined String: Hello! World
```

Example 4: Java compare two strings

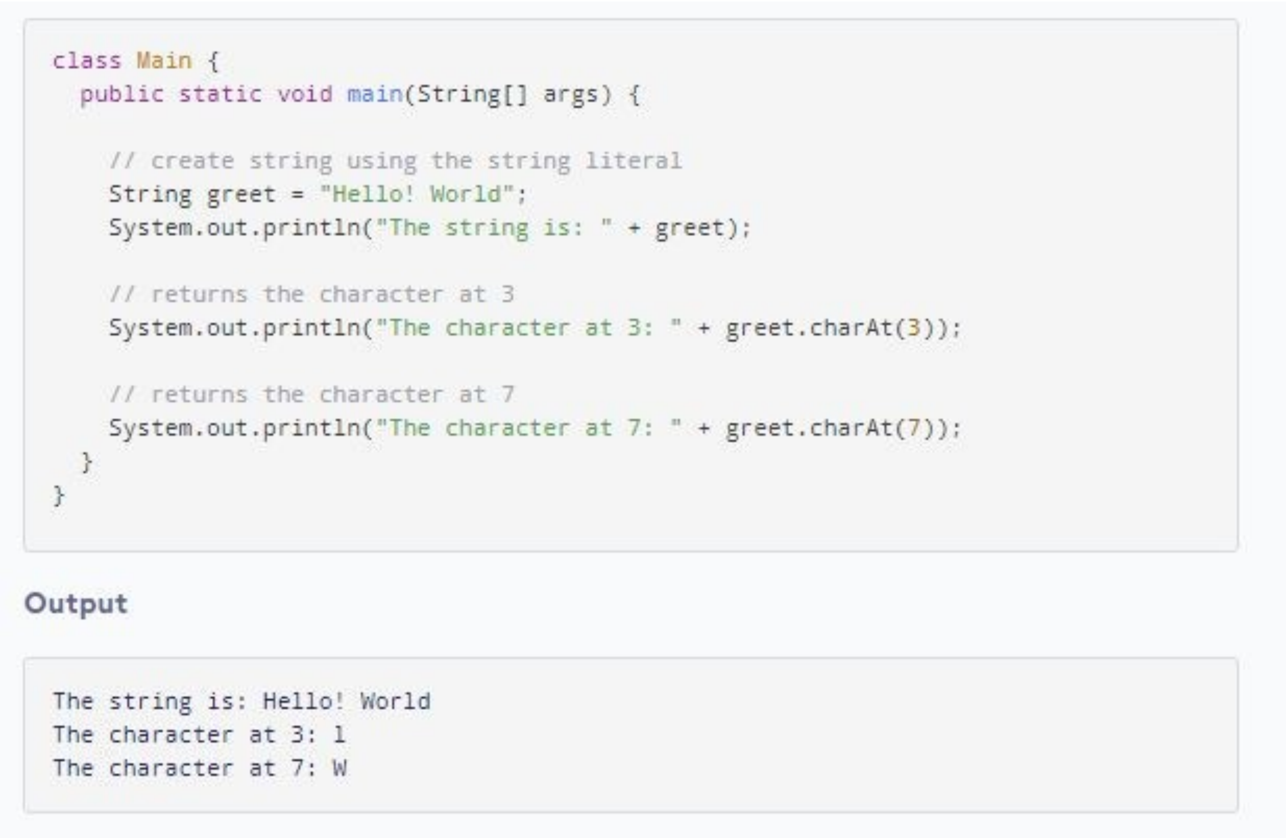


```java
class Main {
  public static void main(String[] args) {

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

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

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

Output

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

Example 5: Java get characters from a string



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

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

    // returns the character at 3
    System.out.println("The character at 3: " + greet.charAt(3));

    // returns the character at 7
    System.out.println("The character at 7: " + greet.charAt(7));
  }
}
```

Output

```
The string is: Hello! World
The character at 3: l
The character at 7: W
```

StringBuilder CLASS

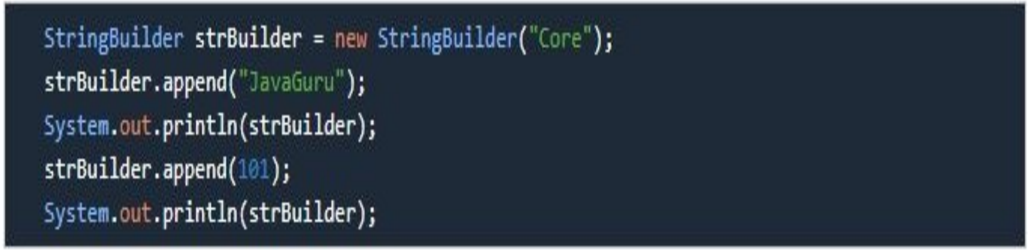
- ❖ StringBuilder objects are like String objects, except that they can be modified. Hence Java StringBuilder class is also used to create mutable (modifiable) string object. StringBuilder is same as StringBuffer except for one important difference. StringBuilder is not synchronized, which means it is not thread safe. At any point, the length and content of the sequence can be changed through method invocations.
- ❖ StringBuilder class provides an API compatible with StringBuffer, but with no guarantee of synchronization. This class is designed for use as a drop-in replacement for StringBuffer in places where the string buffer was being used by a single thread. Where possible, it is recommended that this class be used in preference to StringBuffer as it will be faster under most implementations.
- ❖ Instances of StringBuilder are not safe for use by multiple threads. If such synchronization is required then it is recommended that StringBuffer be used.

CONSTRUCTOR’S OF STRINGBUILDER CLASS

- **StringBuilder ( ) :** Constructs a string builder with no characters in it and an initial capacity of 16 characters.
- **StringBuilder (int capacity) :** Constructs a string builder with no characters in it and an initial capacity specified by the capacity argument.
- **StringBuilder (String str) :** Constructs a string builder initialized to the contents of the specified string. The initial capacity of the string builder is 16 plus the length of the string argument.

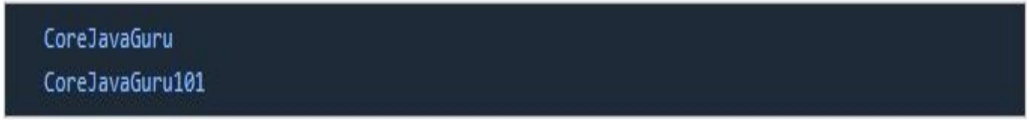
APPEND()

- The append() method concatenates the given argument(string representation) to the end of the invoking StringBuilder object. StringBuilder class has several overloaded append() method. Few are:
  - StringBuilder append(String str)
  - StringBuilder append(int n)
  - StringBuilder append(Object obj)



```
StringBuilder strBuilder = new StringBuilder("Core");
strBuilder.append("JavaGuru");
System.out.println(strBuilder);
strBuilder.append(101);
System.out.println(strBuilder);
```

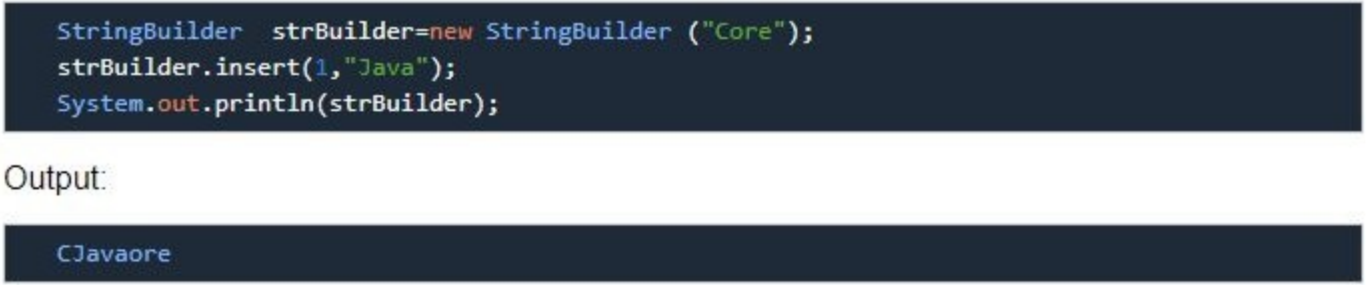
Output:



```
CoreJavaGuru
CoreJavaGuru101
```

INSERT()

❖ The insert() method inserts the given argument(string representation) into the invoking StringBuilder object at the given position.



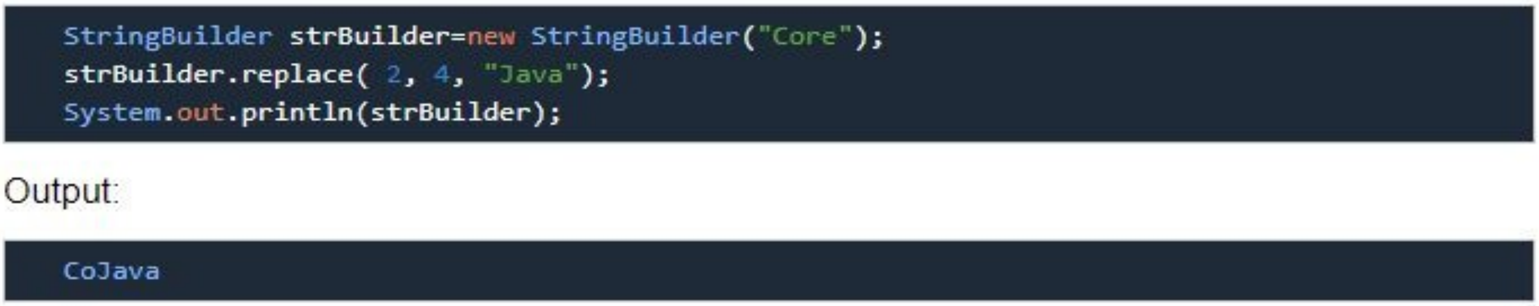
```
StringBuilder  strBuilder=new StringBuilder ("Core");
strBuilder.insert(1,"Java");
System.out.println(strBuilder);
```

Output:

```
CJavaore
```

REPLACE

❖ The replace() method replaces the string from specified start index to the end index.



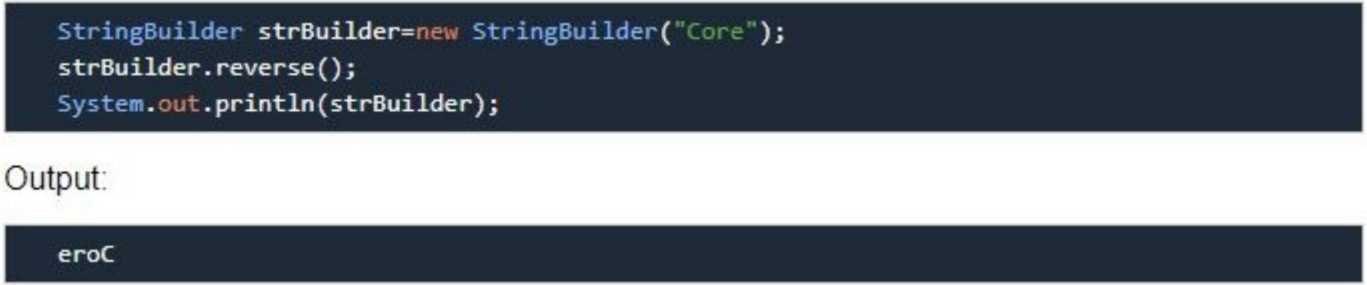
```
StringBuilder strBuilder=new StringBuilder("Core");
strBuilder.replace( 2, 4, "Java");
System.out.println(strBuilder);
```

Output:

```
CoJava
```

REVERSE()

- This method reverses the characters within a StringBuilder object.



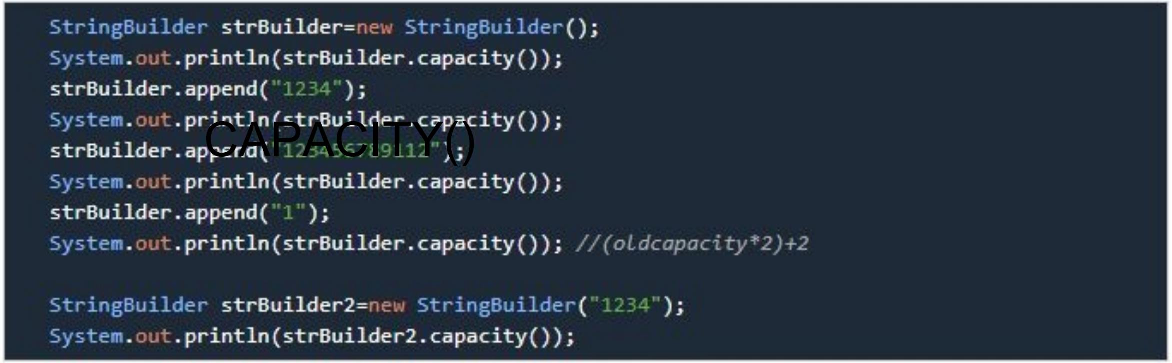
```java
StringBuilder strBuilder=new StringBuilder("Core");
strBuilder.reverse();
System.out.println(strBuilder);
```

Output:

```
eroC
```

CAPACITY()

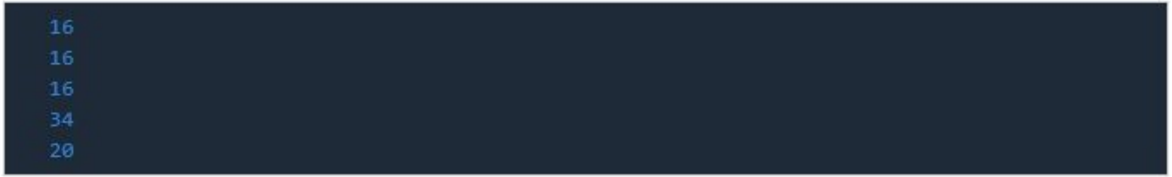
❖ The capacity() method returns the current capacity of StringBuilder object. The capacity is the amount of storage available for newly inserted characters, beyond which an allocation will occur



```java
StringBuilder strBuilder=new StringBuilder();
System.out.println(strBuilder.capacity());
strBuilder.append("1234");
System.out.println(strBuilder.capacity());
strBuilder.append("123456789112");
System.out.println(strBuilder.capacity());
strBuilder.append("1");
System.out.println(strBuilder.capacity()); //(oldcapacity*2)+2

StringBuilder strBuilder2=new StringBuilder("1234");
System.out.println(strBuilder2.capacity());
```

CAPACITY()

Output:



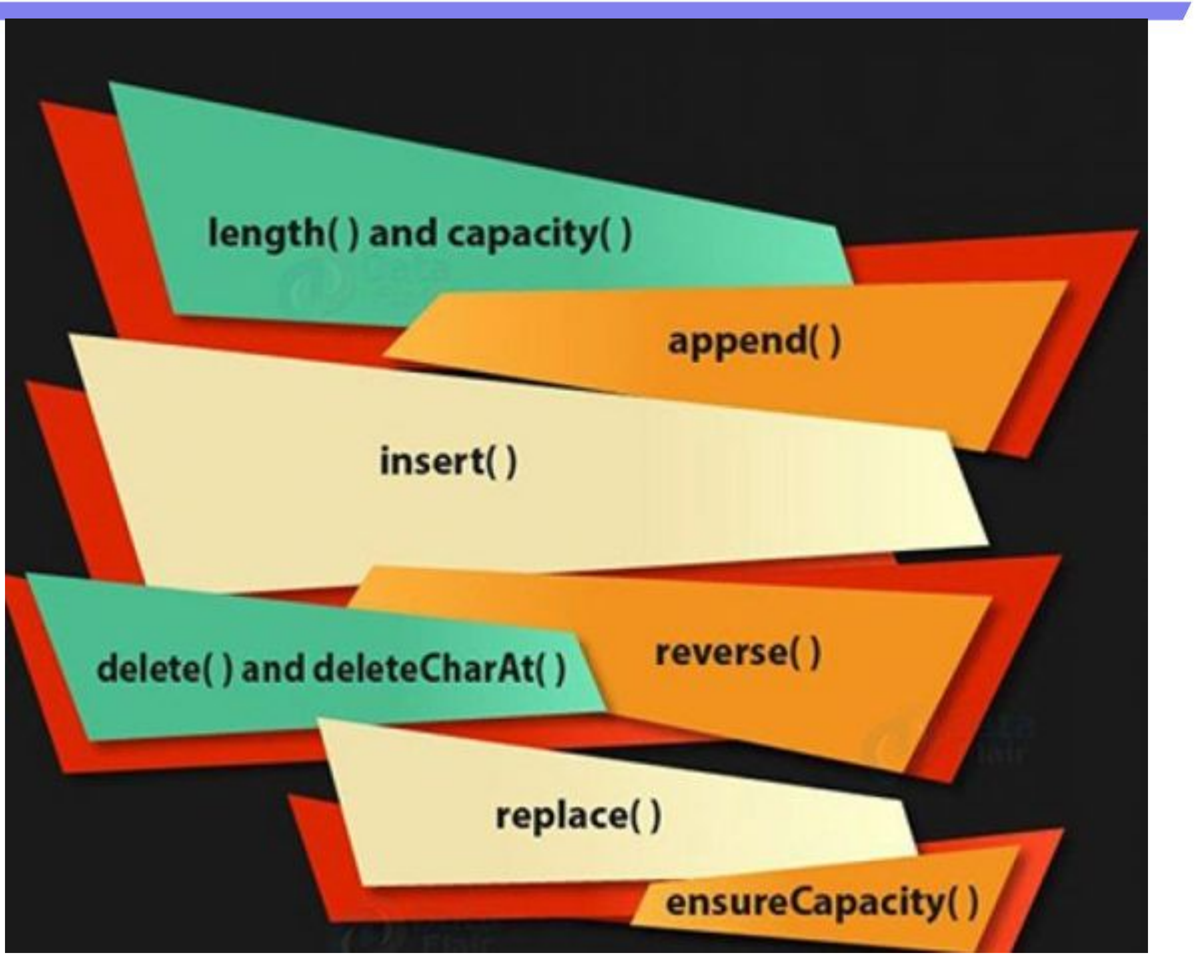
```
16
16
16
34
20
```

STRINGBUFFER CLASS

- ❖ Java StringBuffer class is used to create mutable (modifiable) string object. A string buffer is like a String, but can be modified.
- ❖ As we know that String objects are immutable, so if we do a lot of modifications to String objects, we may end up with a memory leak. To overcome this we use StringBuffer class.
- ❖ StringBuffer class represents growable and writable character sequence. It is also thread-safe i.e. multiple threads cannot access it simultaneously.
- ❖ Every string buffer has a capacity. As long as the length of the character sequence contained in the string buffer does not exceed the capacity, it is not necessary to allocate a new internal buffer array. If the internal buffer overflows, it is automatically made large

CONSTRUCTOR OF STRINGBUFFER CLASS

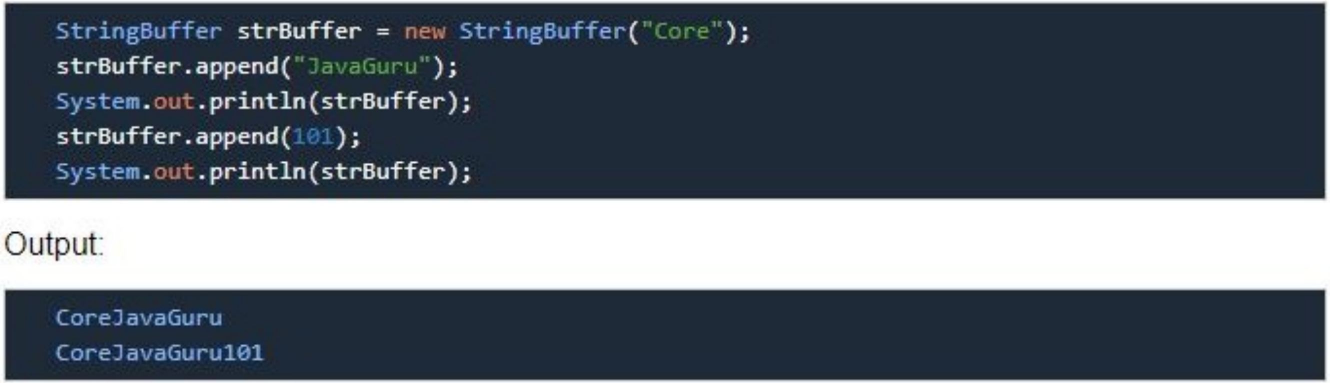
- **StringBuffer ( ) :** Creates an empty string buffer with the initial capacity of 16.
- **StringBuffer (int capacity) :** Creates an empty string buffer with the specified capacity as length.
- **StringBuffer (String str) :** Creates a string buffer initialized to the contents of the specified string.
- **StringBuffer (charSequence[] ch) :** Creates a string buffer that contains the same characters as the specified CharSequence.


length( ) and capacity( )
append( )
insert( )
delete( ) and deleteCharAt( )
reverse( )
replace( )
ensureCapacity( )

APPEND()

- The append() method concatenates the given argument(string representation) to the end of the invoking StringBuffer object. StringBuffer class has several overloaded append() method.

  - StringBuffer append(String str)
  - StringBuffer append(int n)
  - StringBuffer append(Object obj)



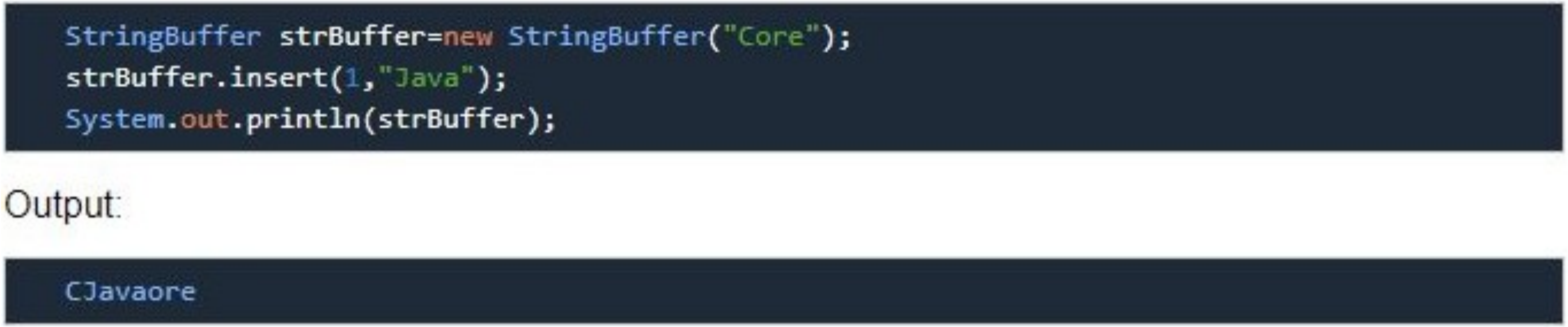
```
StringBuffer strBuffer = new StringBuffer("Core");
strBuffer.append("JavaGuru");
System.out.println(strBuffer);
strBuffer.append(101);
System.out.println(strBuffer);
```

Output:

```
CoreJavaGuru
CoreJavaGuru101
```

INSERT()

- The insert() method inserts the given argument(string representation) into the invoking StringBuffer object at the given position.



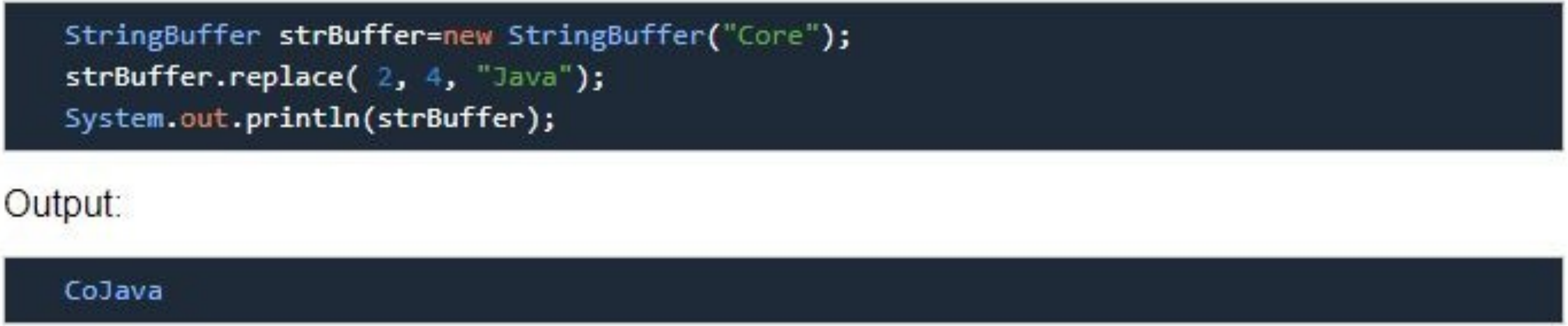
```
StringBuffer strBuffer=new StringBuffer("Core");
strBuffer.insert(1,"Java");
System.out.println(strBuffer);
```

Output:

```
CJavaore
```

REPLACE()

- The replace() method replaces the string from specified start index to the end index.



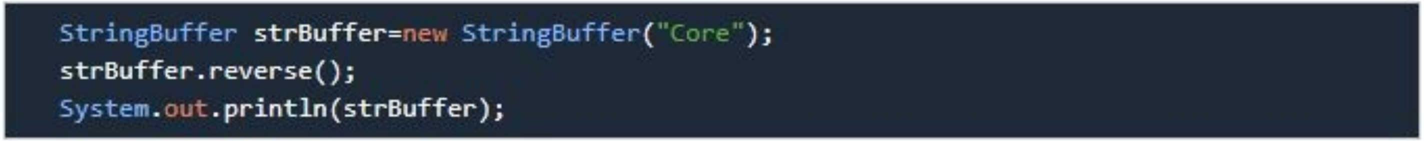
```
StringBuffer strBuffer=new StringBuffer("Core");
strBuffer.replace( 2, 4, "Java");
System.out.println(strBuffer);
```

Output:

```
CoJava
```

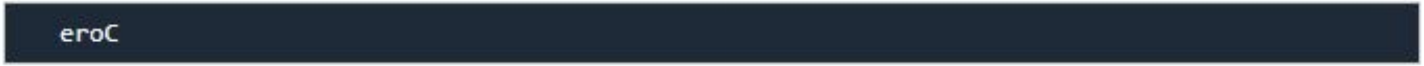
REVERSE()

- This method reverses the characters within a StringBuffer object.



```
StringBuffer strBuffer=new StringBuffer("Core");
strBuffer.reverse();
System.out.println(strBuffer);
```

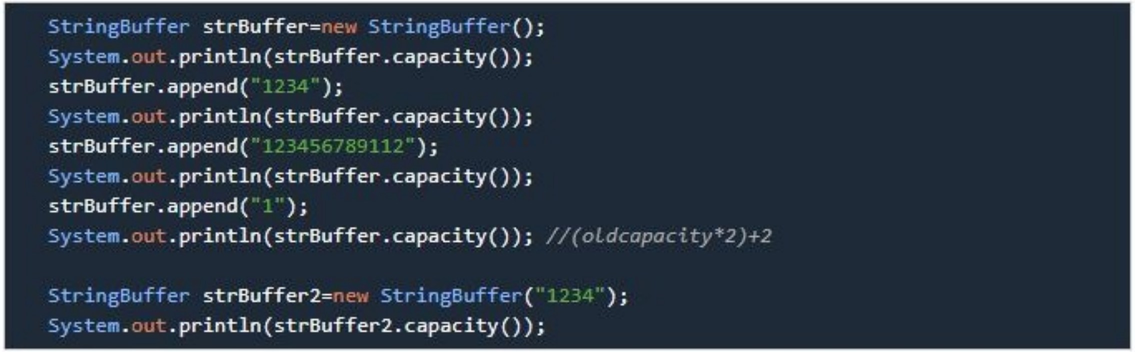
Output:



```
eroC
```

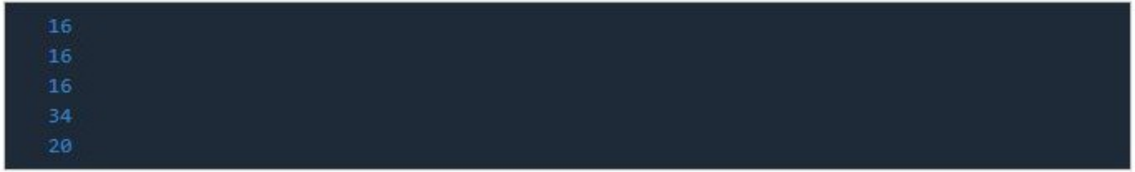
CAPACITY()

- The capacity() method returns the current capacity of StringBuffer object. The capacity is the amount of storage available for newly inserted characters, beyond which an allocation will occur.

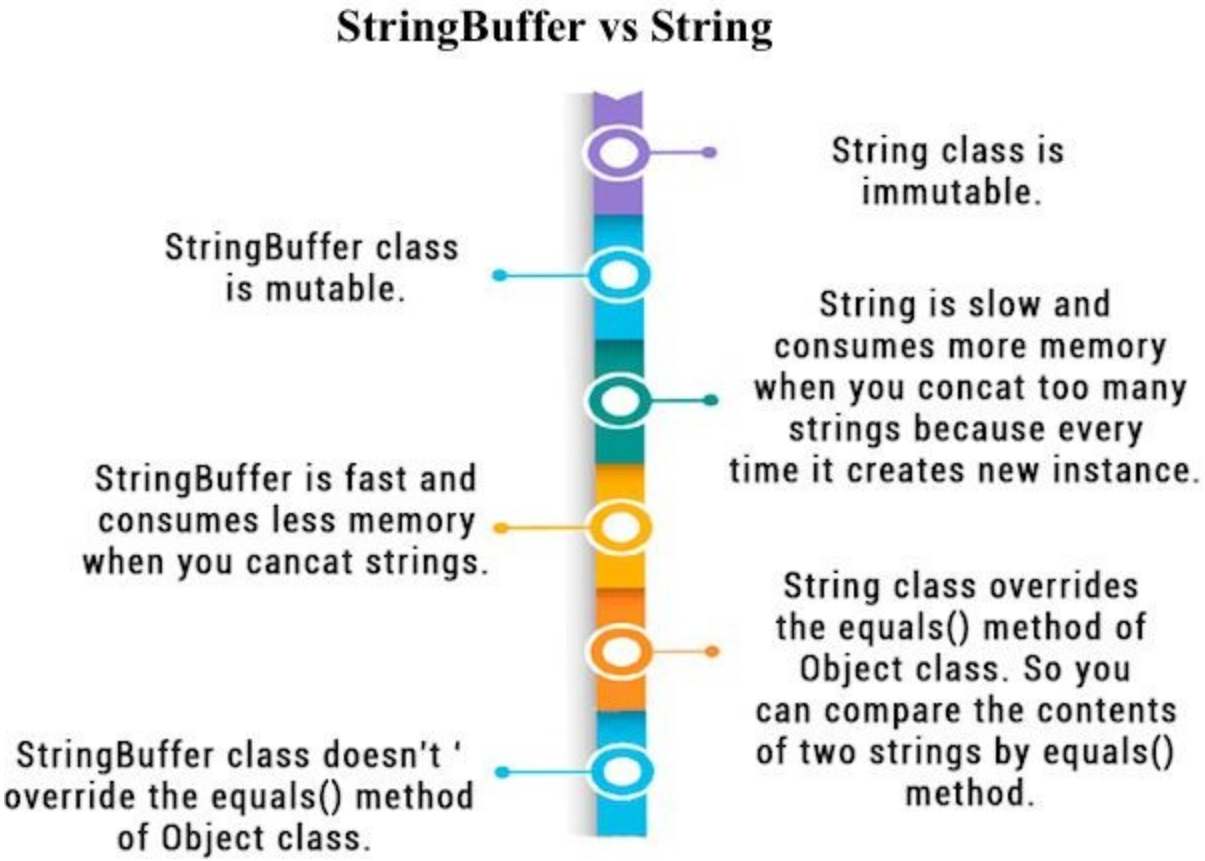


```
StringBuffer strBuffer=new StringBuffer();
System.out.println(strBuffer.capacity());
strBuffer.append("1234");
System.out.println(strBuffer.capacity());
strBuffer.append("123456789112");
System.out.println(strBuffer.capacity());
strBuffer.append("1");
System.out.println(strBuffer.capacity()); //(oldcapacity*2)+2

StringBuffer strBuffer2=new StringBuffer("1234");
System.out.println(strBuffer2.capacity());
```

Output:



```
16
16
16
34
20
```



StringBuffer vs String

String class is immutable.

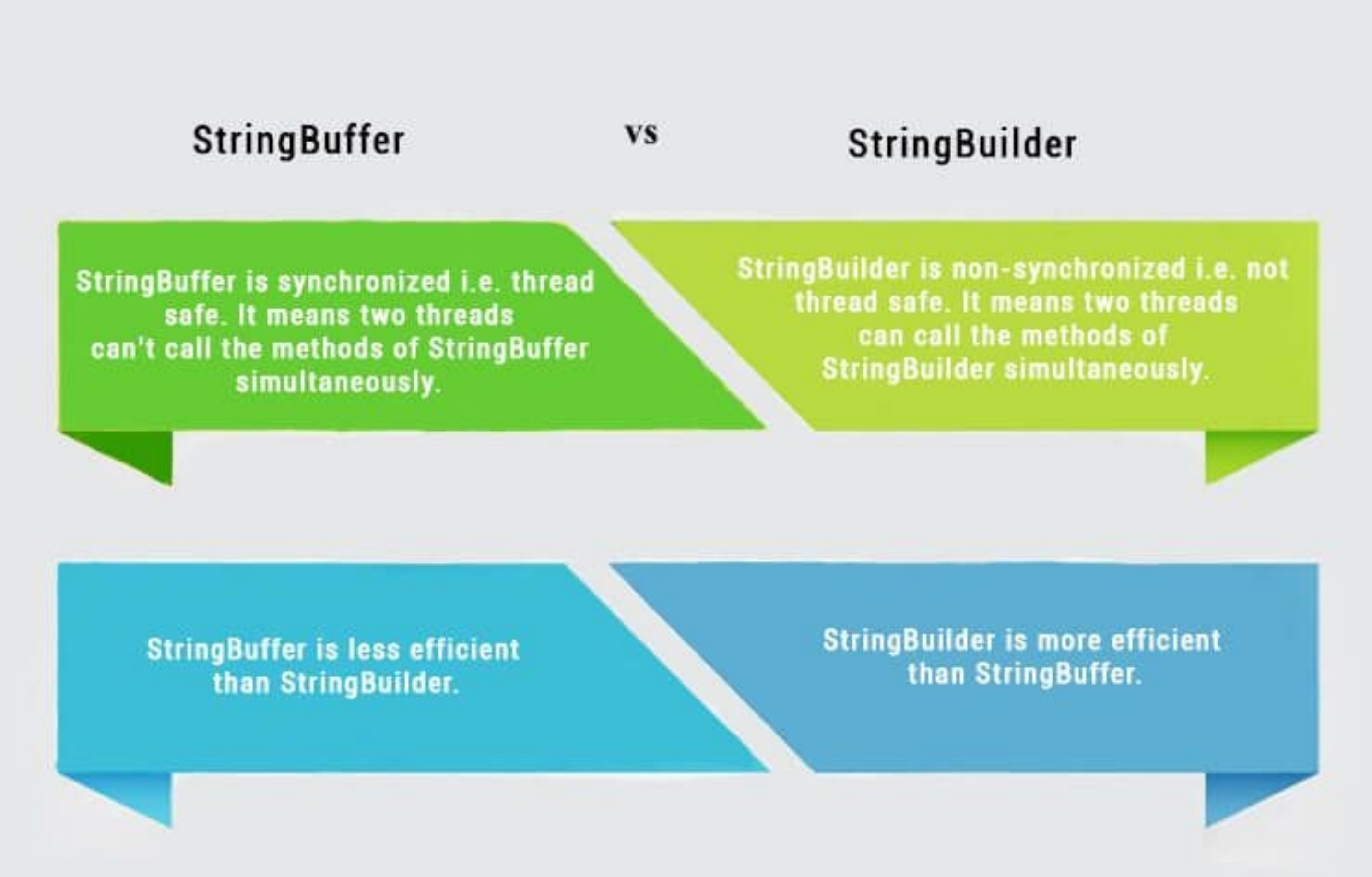
StringBuffer class is mutable.

String is slow and consumes more memory when you concat too many strings because every time it creates new instance.

StringBuffer is fast and consumes less memory when you cancat strings.

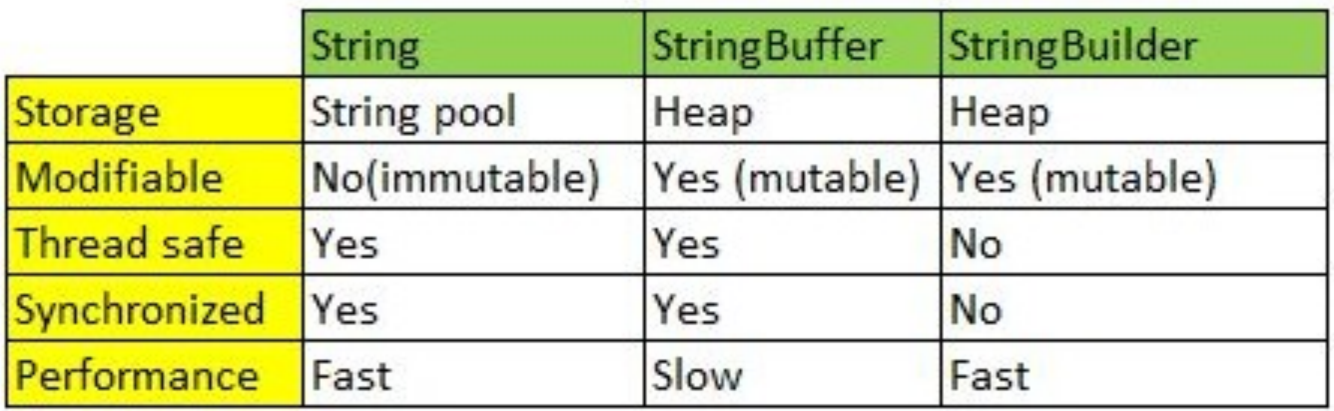
String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.

StringBuffer class doesn't ' override the equals() method of Object class.



StringBuffer vs StringBuilder

StringBuffer	StringBuilder
StringBuffer is synchronized i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is non-synchronized i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
StringBuffer is less efficient than StringBuilder.	StringBuilder is more efficient than StringBuffer.

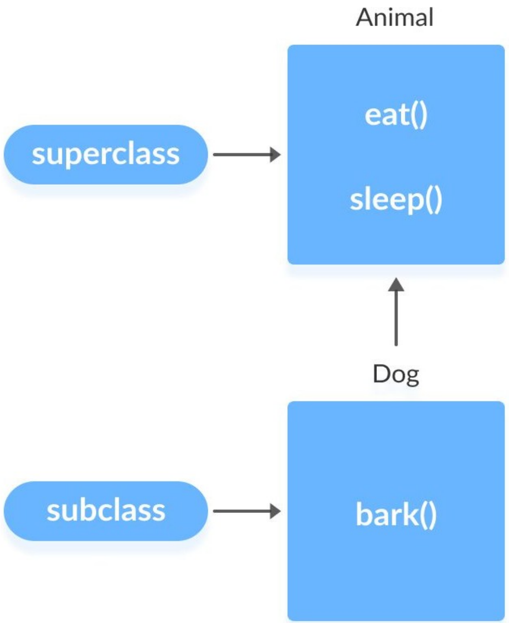


	String	StringBuffer	StringBuilder
Storage	String pool	Heap	Heap
Modifiable	No(immutable)	Yes (mutable)	Yes (mutable)
Thread safe	Yes	Yes	No
Synchronized	Yes	Yes	No
Performance	Fast	Slow	Fast

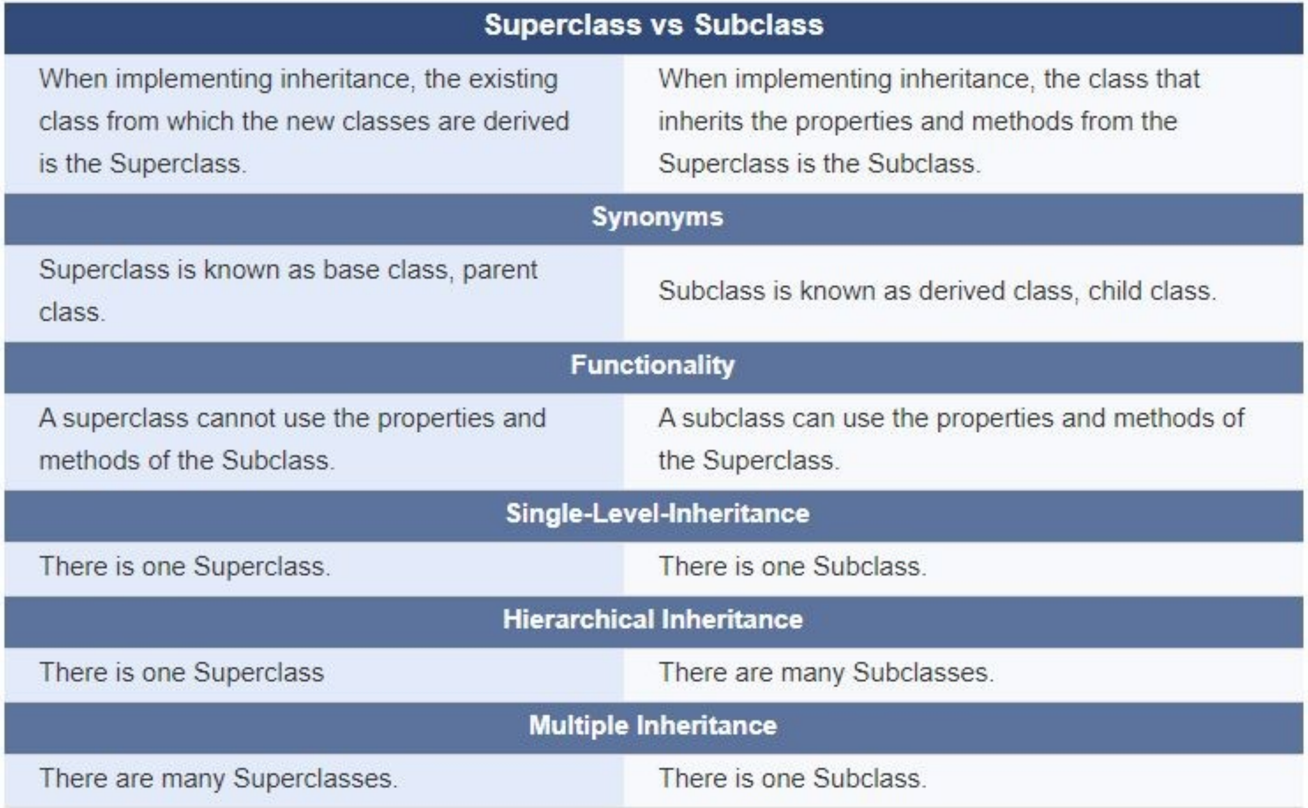
INHERITANCE AND POLYMORPHISM

SUPERCLASS AND SUBCLASS

- **Java Inheritance** (**Subclass** and **Superclass**) In **Java**, it is possible to inherit attributes and methods from one class to another. ... **subclass** (child) - the class that inherits from another class. **superclass** (parent) - the class being inherited from.

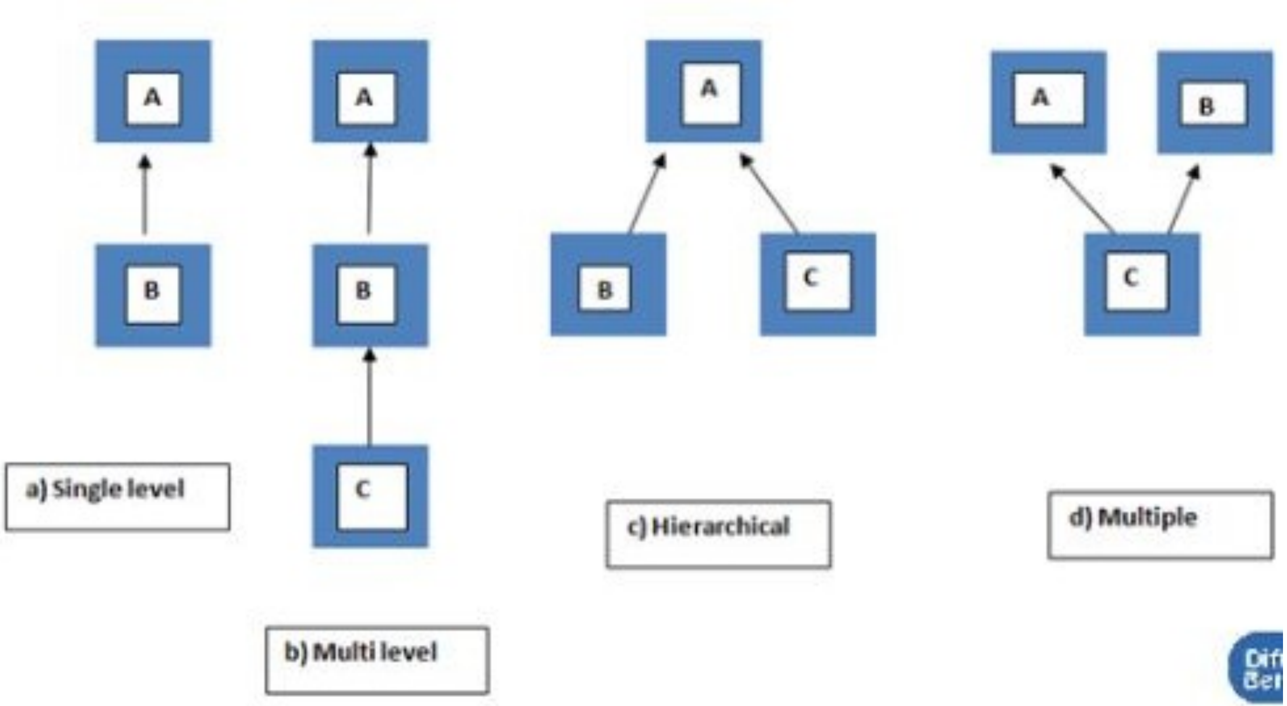


DIFFERENCE BETWEEN SUPERCLASS & SUBCLASS



Superclass vs Subclass	
When implementing inheritance, the existing class from which the new classes are derived is the Superclass.	When implementing inheritance, the class that inherits the properties and methods from the Superclass is the Subclass.
Synonyms	
Superclass is known as base class, parent class.	Subclass is known as derived class, child class.
Functionality	
A superclass cannot use the properties and methods of the Subclass.	A subclass can use the properties and methods of the Superclass.
Single-Level-Inheritance	
There is one Superclass.	There is one Subclass.
Hierarchical Inheritance	
There is one Superclass	There are many Subclasses.
Multiple Inheritance	
There are many Superclasses.	There is one Subclass.

TYPES OF INHERITANCE

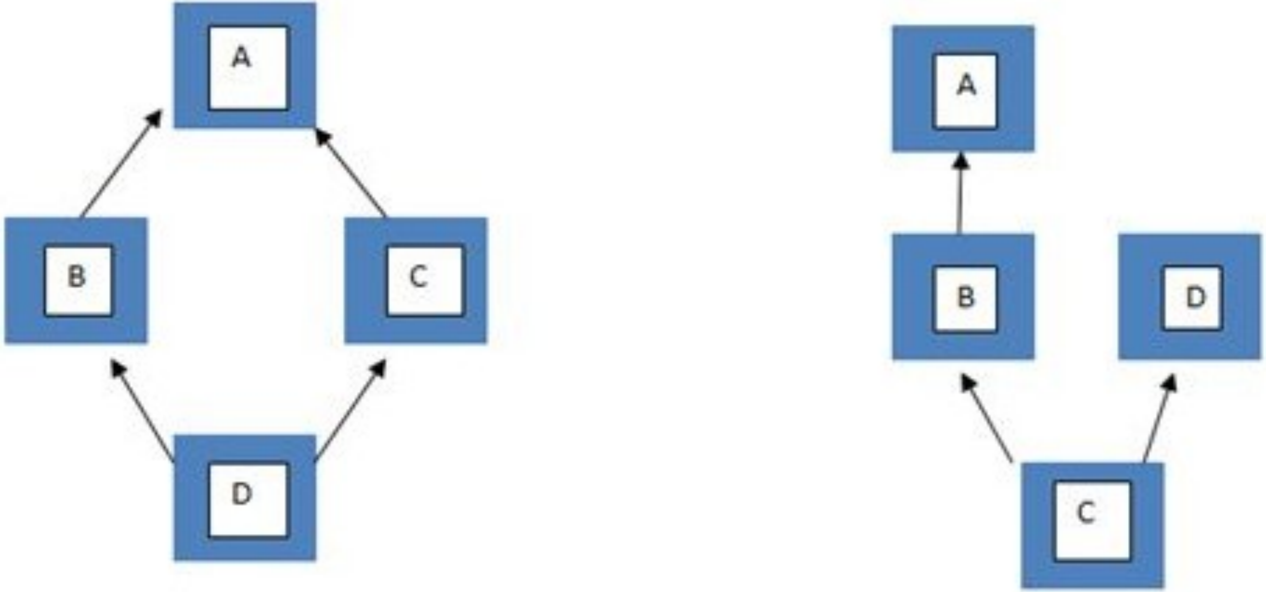


a) Single level

b) Multi level

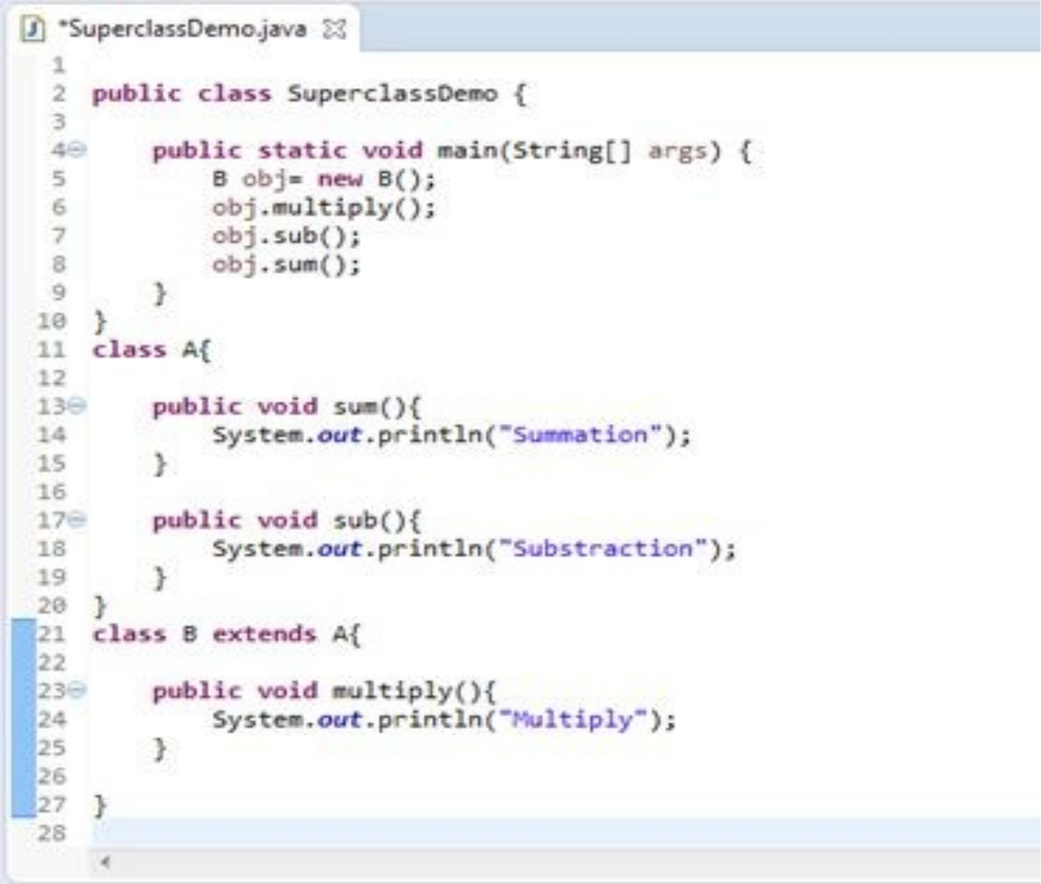
c) Hierarchical

d) Multiple



e) Hybrid

- According to the above diagrams, Superclasses varies from each inheritance type. In single-level inheritance, A is the Superclass. In Multilevel inheritance, A is the Superclass for B and B is the Superclass for C. In Hierarchical Inheritance A is the Superclass for both B and C. In multiple inheritances both A and B are Superclasses for C.
- Hybrid inheritance is a combination of multi-level and multiple inheritances. In the left-hand side diagram, A is the Superclass for B, C and B, C are the Superclasses for D. In the right-hand side diagram, A is the Superclass for B and B, D are Superclasses for C.



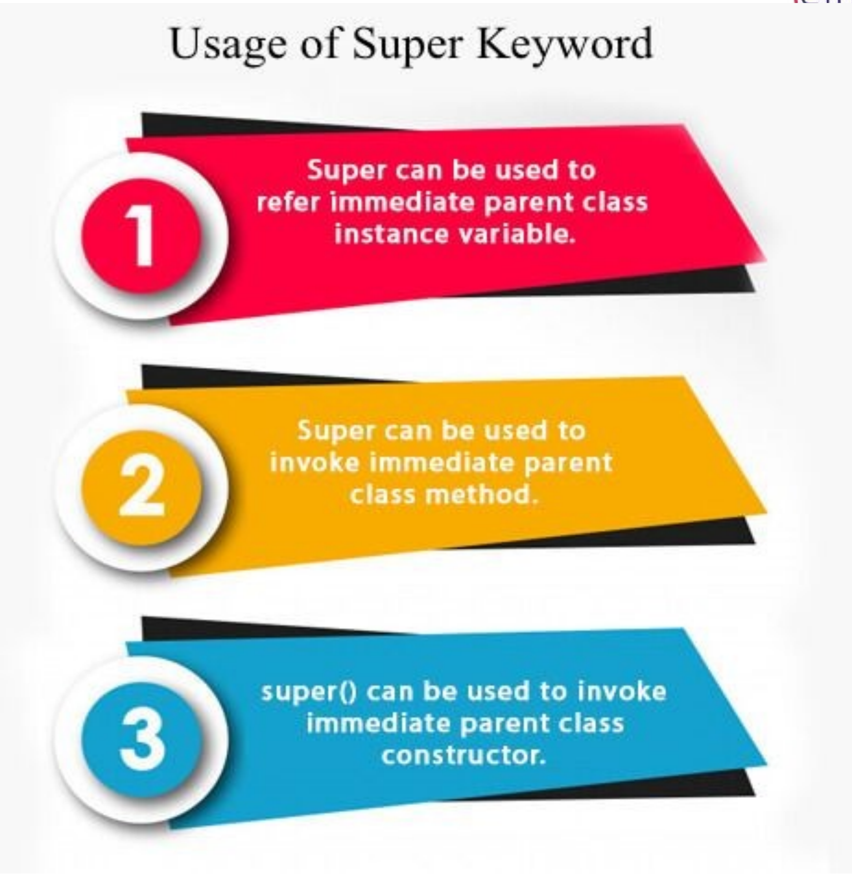
*SuperclassDemo.java

```java

public class SuperclassDemo {

    public static void main(String[] args) {
        B obj= new B();
        obj.multiply();
        obj.sub();
        obj.sum();
    }
}
class A{

    public void sum(){
        System.out.println("Summation");
    }

    public void sub(){
        System.out.println("Substraction");
    }
}
class B extends A{

    public void multiply(){
        System.out.println("Multiply");
    }

}

```

- According to the above program, class A have sum() and sub() methods. Class B has multiply() method. Class B is extending class A. Therefore, properties and methods of class A are accessible by class B. Therefore, class A is the Superclass. The reference type of class B is taken to create the object. So, all methods such as sum(), sub() and multiply() are accessible by the object. If Superclass reference type is used for object creation, the members of class B cannot be accessible. e.g. A obj = new B(); Therefore, Superclass reference cannot call the method multiply() because that method belongs to class B.

SUPER KEYWORD

- The **super keyword in Java** is a reference variable which is used to refer immediate parent class object. Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by **super** reference variable.

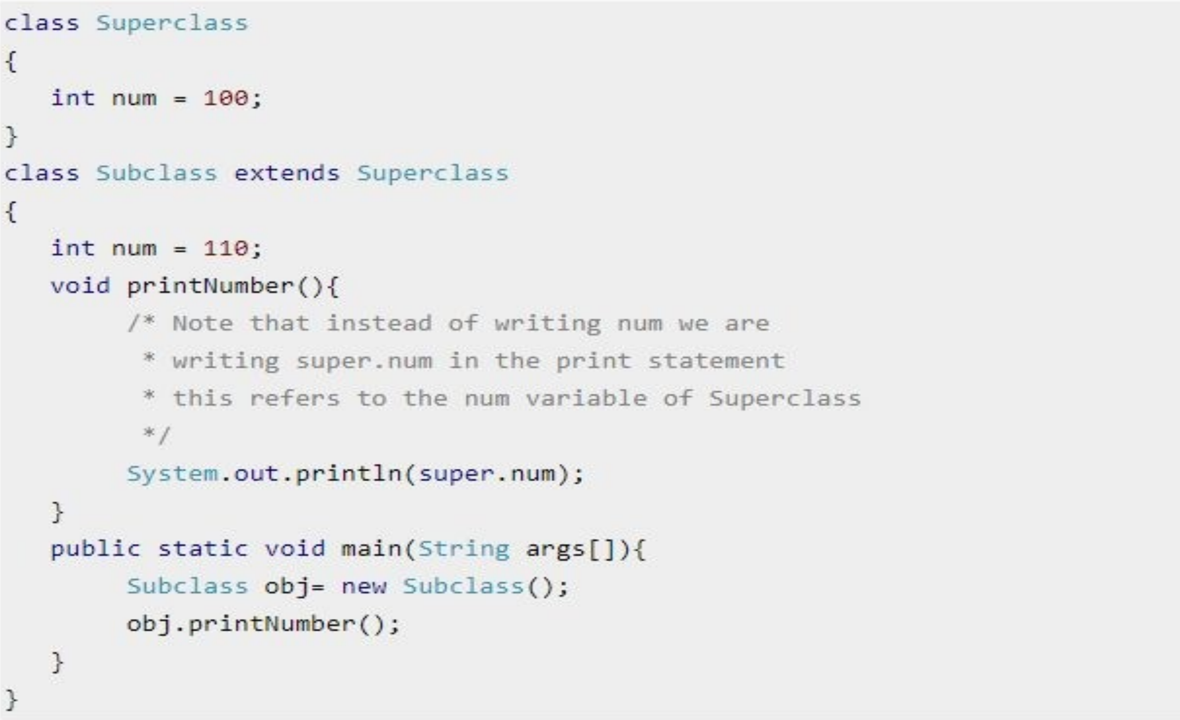


Usage of Super Keyword

1 Super can be used to refer immediate parent class instance variable.

2 Super can be used to invoke immediate parent class method.

3 super() can be used to invoke immediate parent class constructor.



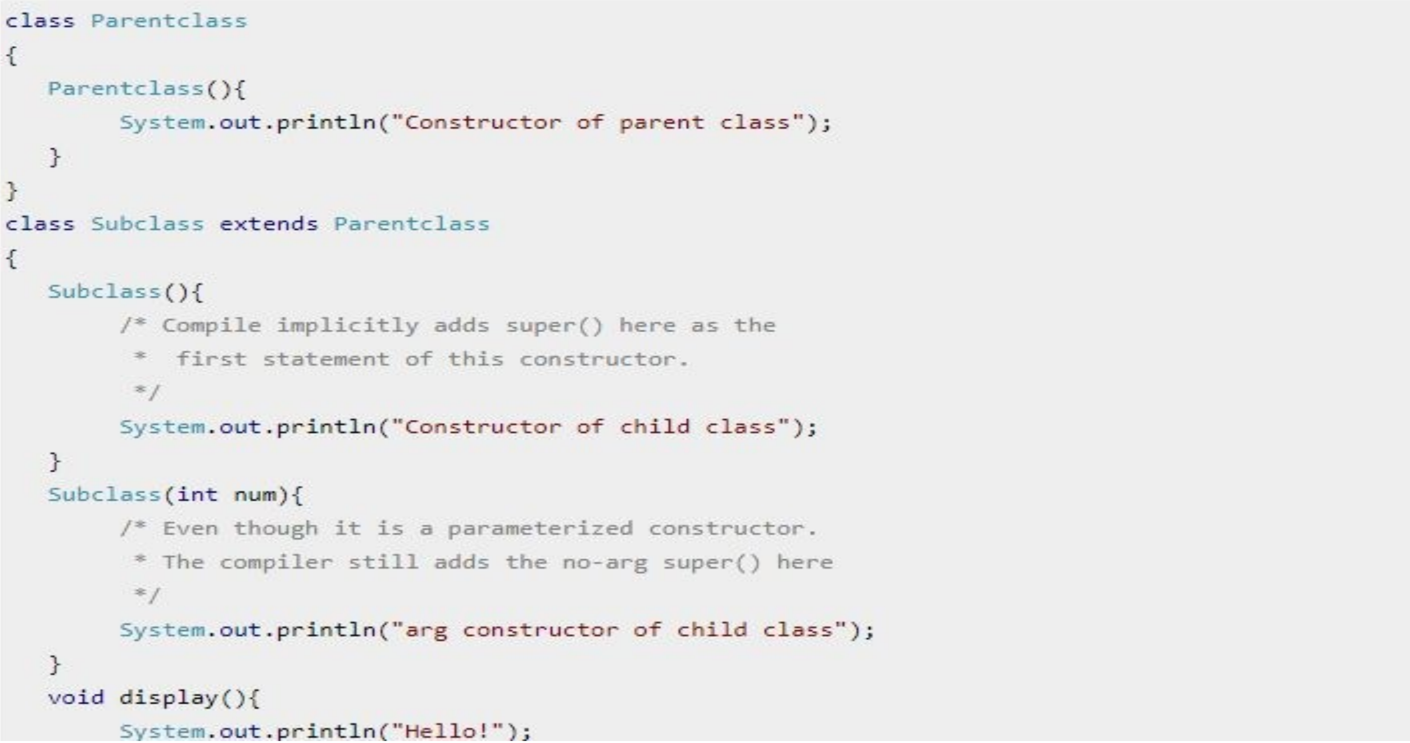
```
class Superclass
{
   int num = 100;
}
class Subclass extends Superclass
{
   int num = 110;
   void printNumber(){
        /* Note that instead of writing num we are
         * writing super.num in the print statement
         * this refers to the num variable of Superclass
         */
        System.out.println(super.num);
   }
   public static void main(String args[]){
        Subclass obj= new Subclass();
        obj.printNumber();
   }
}
```

Output:

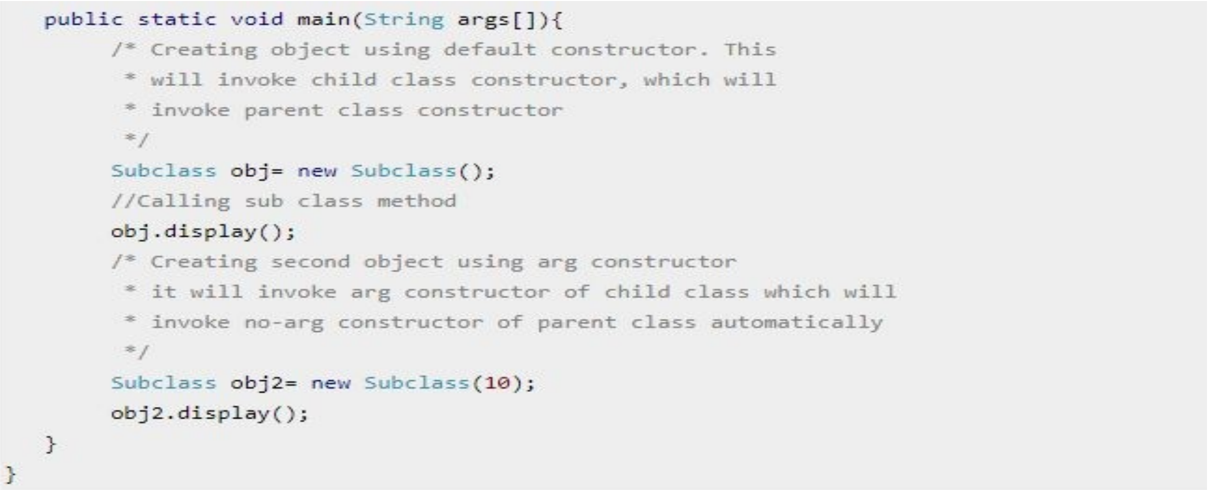
100

Use of super keyword to invoke constructor of parent class

❖ When we create the object of sub class, the new keyword invokes the constructor of child class, which implicitly invokes the constructor of parent class. So the order to execution when we create the object of child class is: parent class constructor is executed first and then the child class constructor is executed. It happens because compiler itself adds super()(this invokes the no-arg constructor of parent class) as the first statement in the constructor of child class.

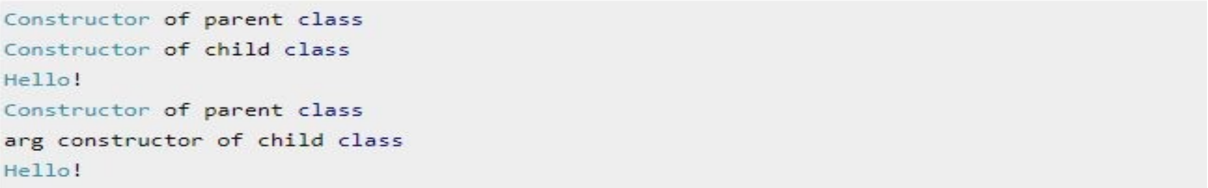


```
class Parentclass
{
   Parentclass(){
        System.out.println("Constructor of parent class");
   }
}
class Subclass extends Parentclass
{
   Subclass(){
        /* Compile implicitly adds super() here as the
         *  first statement of this constructor.
         */
        System.out.println("Constructor of child class");
   }
   Subclass(int num){
        /* Even though it is a parameterized constructor.
         * The compiler still adds the no-arg super() here
         */
        System.out.println("arg constructor of child class");
   }
   void display(){
        System.out.println("Hello!");
```



```
   public static void main(String args[]){
        /* Creating object using default constructor. This
         * will invoke child class constructor, which will
         * invoke parent class constructor
         */
        Subclass obj= new Subclass();
        //Calling sub class method
        obj.display();
        /* Creating second object using arg constructor
         * it will invoke arg constructor of child class which will
         * invoke no-arg constructor of parent class automatically
         */
        Subclass obj2= new Subclass(10);
        obj2.display();
   }
}
```

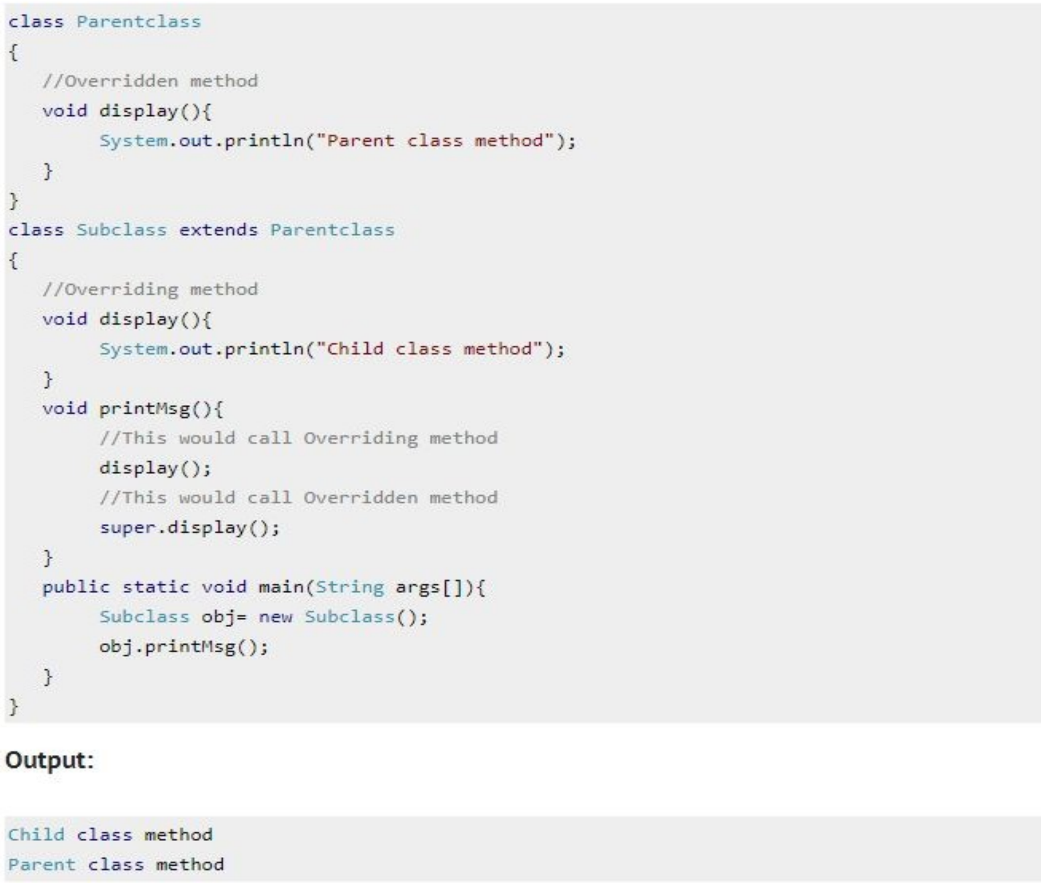
Output:



```
Constructor of parent class
Constructor of child class
Hello!
Constructor of parent class
arg constructor of child class
Hello!
```

How to use super keyword in case of method overriding

- When a child class declares a same method which is already present in the parent class then this is called method overriding. We will learn method overriding in the next tutorials of this series. For now you just need to remember this: When a child class overrides a method of parent class, then the call to the method from child class object always call the child class version of the method. However by using super keyword like this: super.method_name you can call the method of parent class (the method which is overridden). In case of method overriding, these terminologies are used: Overridden method: The method of parent class Overriding method: The method of child class Lets take an example to understand this concept:



```
class Parentclass
{
   //Overridden method
   void display(){
        System.out.println("Parent class method");
   }
}
class Subclass extends Parentclass
{
   //Overriding method
   void display(){
        System.out.println("Child class method");
   }
   void printMsg(){
        //This would call Overriding method
        display();
        //This would call Overridden method
        super.display();
   }
   public static void main(String args[]){
        Subclass obj= new Subclass();
        obj.printMsg();
   }
}
```

Output:

```
Child class method
Parent class method
```

OVERRIDING AND OVERLOADING METHOD

- **Method overriding** is used to provide the specific implementation of the **method** that is already provided by its super class. ... 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.

METHOD OVERLOADING

- Method overloading allows the method to have the same name which differs based on arguments or the argument types. It can be related to compile-time polymorphism.

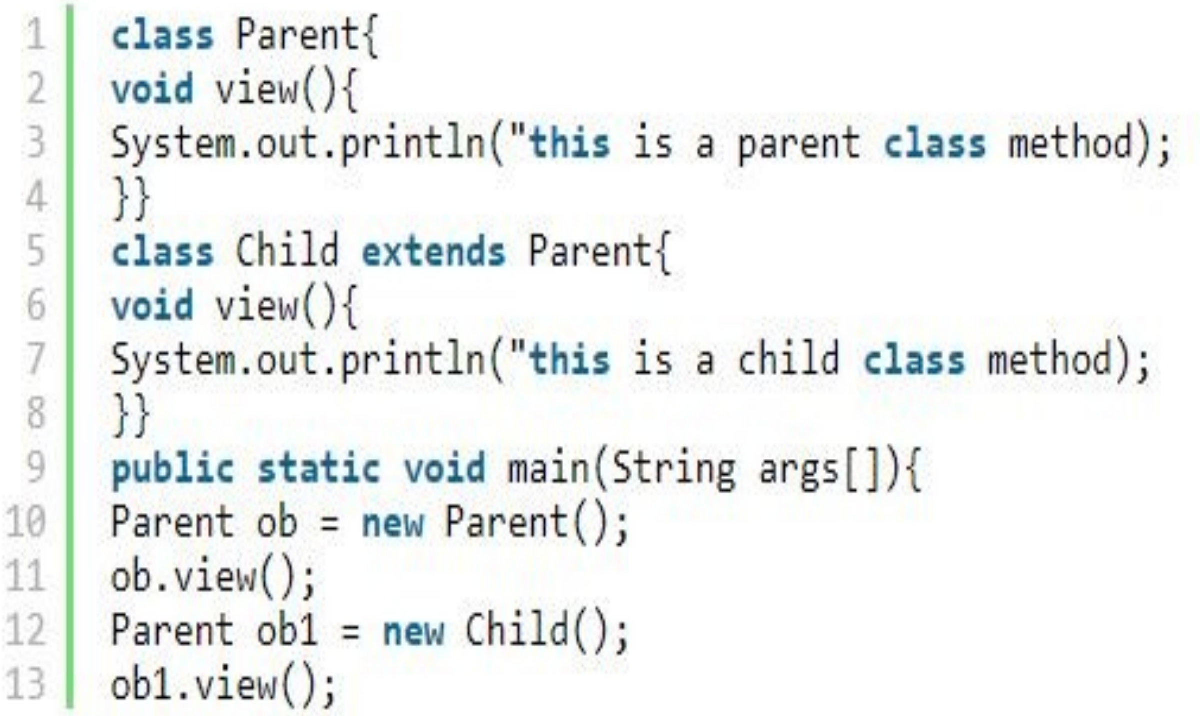
Following are a few pointers we must keep in mind while overloading methods in Java.

- We cannot overload a return type.
- Although we can overload static methods, the arguments or input parameters have to be different.
- We cannot overload two methods if they only differ by a static keyword.
- Like other static methods, the main() method can also be overloaded.

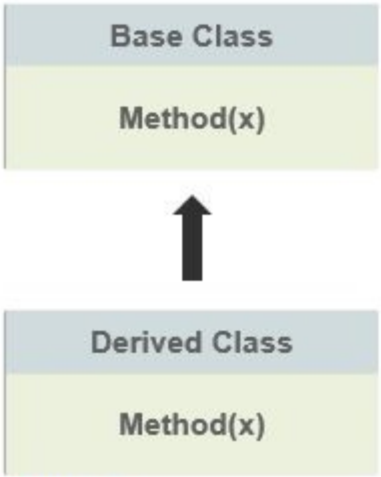
```
public class SimpleOverloading {

    // Method to print a string
    public void printMessage(String message) {
        System.out.println("String message: " + message);
    }

    // Method to print an integer
    public void printMessage(int number) {
        System.out.println("Integer message: " + number);
    }

    // Method to print a double
    public void printMessage(double number) {
        System.out.println("Double message: " + number);
    }

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

        // Calling the overloaded methods
        obj.printMessage("Hello, World!");
        obj.printMessage(100);
        obj.printMessage(99.99);
    }
}
```

METHOD OVERRIDING

- Inheritance in java involves a relationship between parent and child classes. Whenever both the classes contain methods with the same name and arguments or parameters it is certain that one of the methods will override the other method during execution. The method that will be executed depends on the object.
- If the child class object calls the method, the child class method will override the parent class method. Otherwise, if the parent class object calls the method, the parent class method will be executed.



```java
class Parent{
void view(){
System.out.println("this is a parent class method);
}}
class Child extends Parent{
void view(){
System.out.println("this is a child class method);
}}
public static void main(String args[]){
Parent ob = new Parent();
ob.view();
Parent ob1 = new Child();
ob1.view();
```

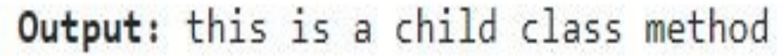


Base Class

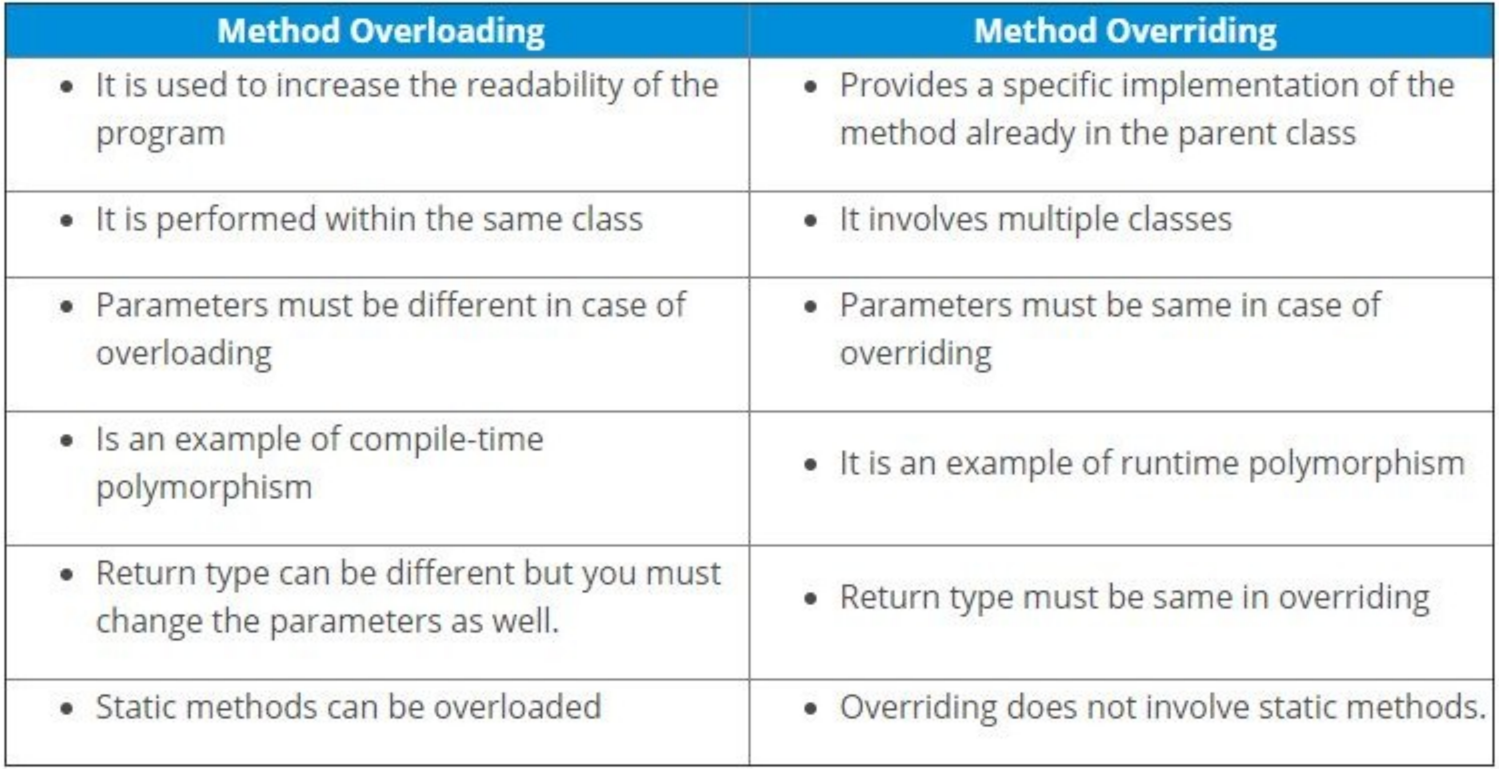
Method(x)

Derived Class

Method(x)



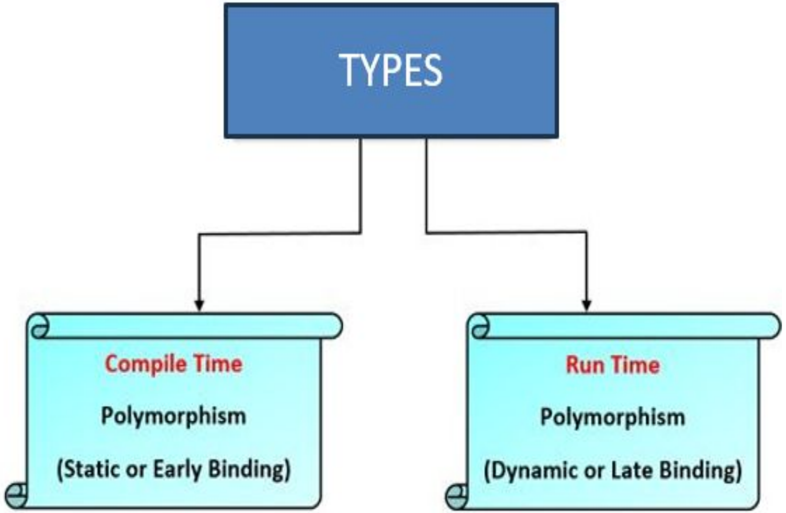
```
Output: this is a child class method
```

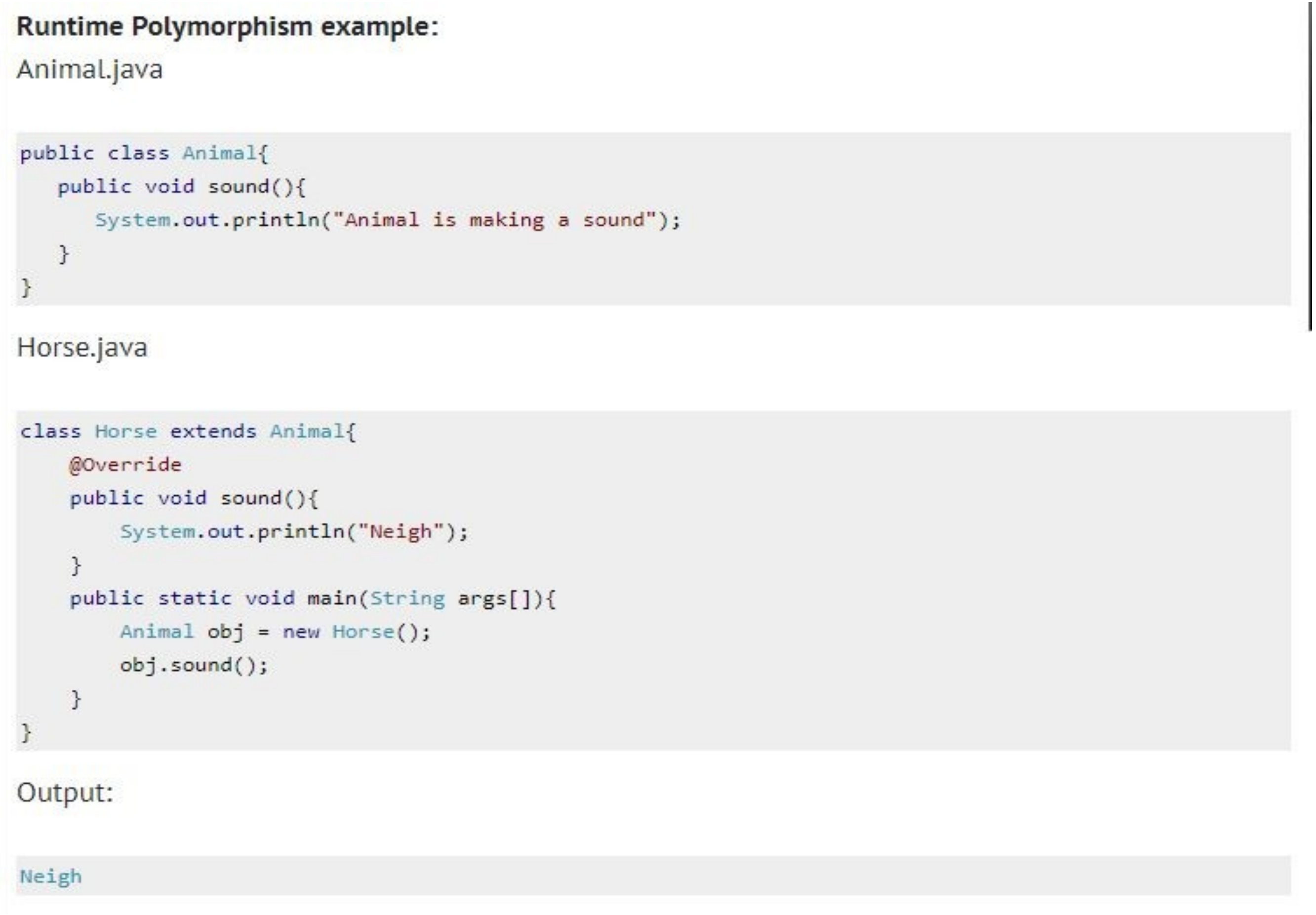


Method Overloading	Method Overriding
It is used to increase the readability of the program	Provides a specific implementation of the method already in the parent class
It is performed within the same class	It involves multiple classes
Parameters must be different in case of overloading	Parameters must be same in case of overriding
Is an example of compile-time polymorphism	It is an example of runtime polymorphism
Return type can be different but you must change the parameters as well.	Return type must be same in overriding
Static methods can be overloaded	Overriding does not involve static methods.

POLYMORPHISM AND DYNAMIC BINDING

- ❖ **Polymorphism in Java** is a concept by which we can perform a single action in different ways. ... So **polymorphism** means many forms. There are two types of **polymorphism in Java**: compile-time **polymorphism** and runtime **polymorphism**. We can perform **polymorphism in java** by method overloading and method overriding.





Runtime Polymorphism example:

Animal.java

```
public class Animal{
   public void sound(){
      System.out.println("Animal is making a sound");
   }
}
```

Horse.java

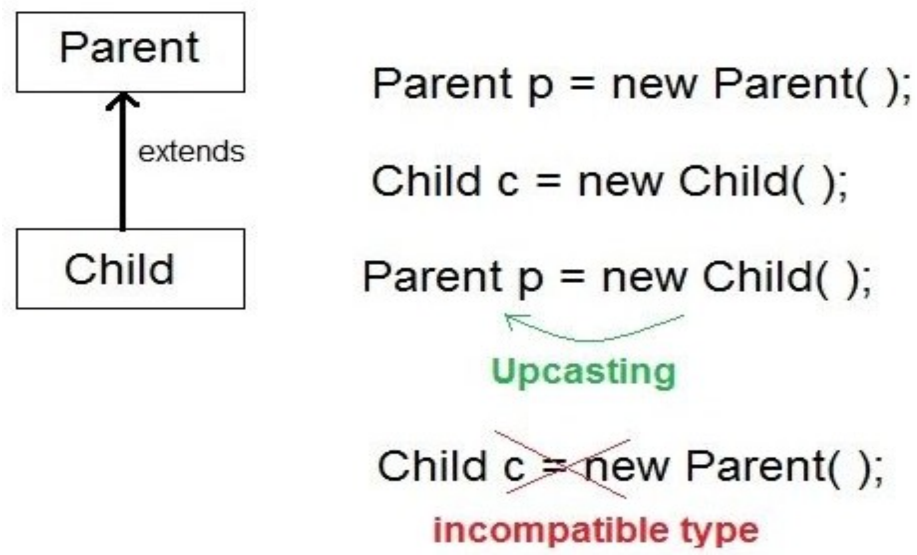
```
class Horse extends Animal{
    @Override
    public void sound(){
        System.out.println("Neigh");
    }
    public static void main(String args[]){
        Animal obj = new Horse();
        obj.sound();
    }
}
```

Output:

```
Neigh
```

DYNAMIC METHOD DISPATCH

- **Dynamic method dispatch** is a mechanism by which a call to an overridden **method** is resolved at runtime. This is how **java** implements runtime polymorphism. When an overridden **method** is called by a reference, **java** determines which version of that **method** to execute based on the type of object it refer to.



```
Parent p = new Parent( );

Child c = new Child( );

Parent p = new Child( );
```

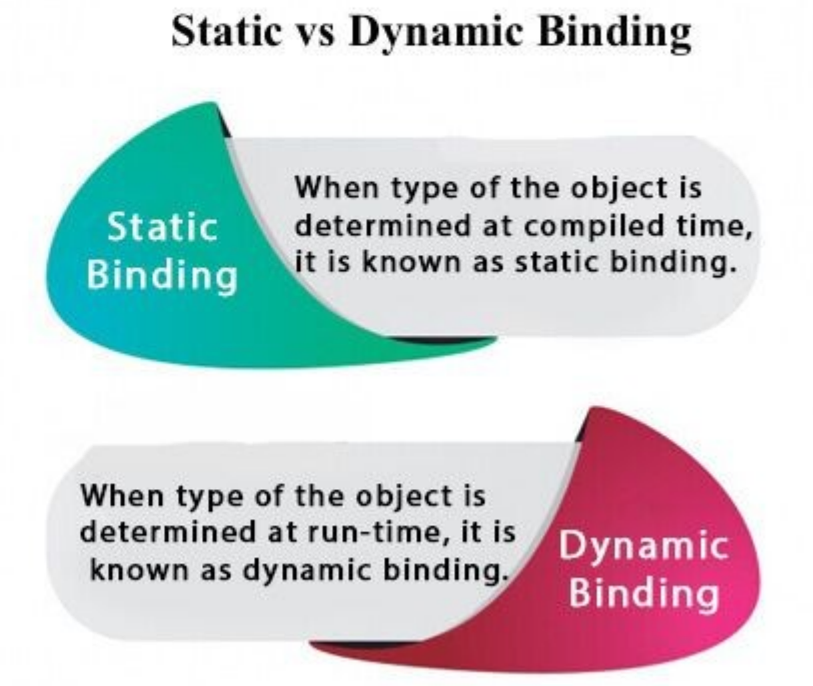
Upcasting

Child c ~~= new~~ Parent( );

incompatible type

DYNAMIC BINDING

When compiler is not able to resolve the call/binding at compile time, such binding is known as Dynamic or late Binding. Method Overriding is a perfect example of dynamic binding as in overriding both parent and child classes have same method and in this case the **type of the object** determines which method is to be executed. The type of object is determined at the run time so this is known as dynamic binding.



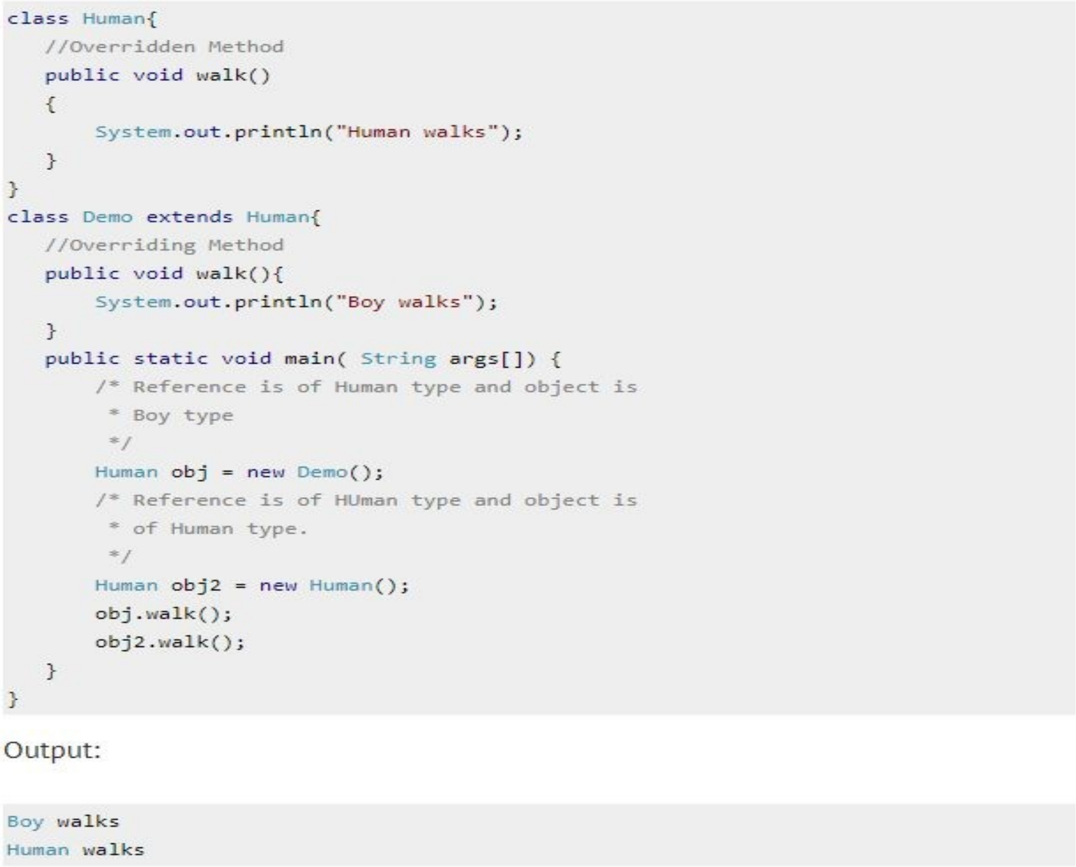
Static vs Dynamic Binding

Static Binding

When type of the object is determined at compiled time, it is known as static binding.

When type of the object is determined at run-time, it is known as dynamic binding.

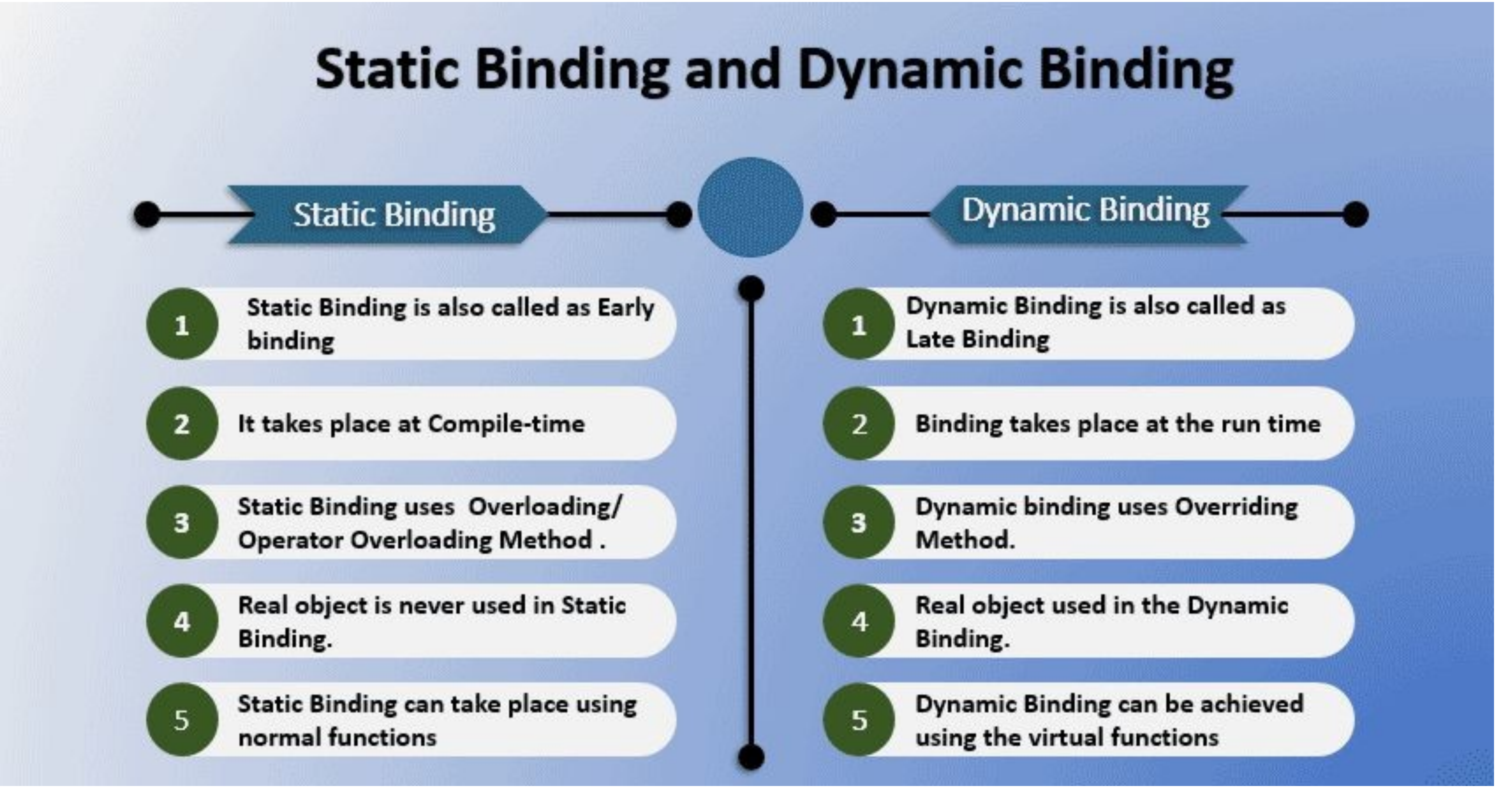
Dynamic Binding



```
class Human{
   //Overridden Method
   public void walk()
   {
       System.out.println("Human walks");
   }
}
class Demo extends Human{
   //Overriding Method
   public void walk(){
       System.out.println("Boy walks");
   }
   public static void main( String args[]) {
       /* Reference is of Human type and object is
        * Boy type
        */
       Human obj = new Demo();
       /* Reference is of HUman type and object is
        * of Human type.
        */
       Human obj2 = new Human();
       obj.walk();
       obj2.walk();
   }
}
```

Output:

```
Boy walks
Human walks
```

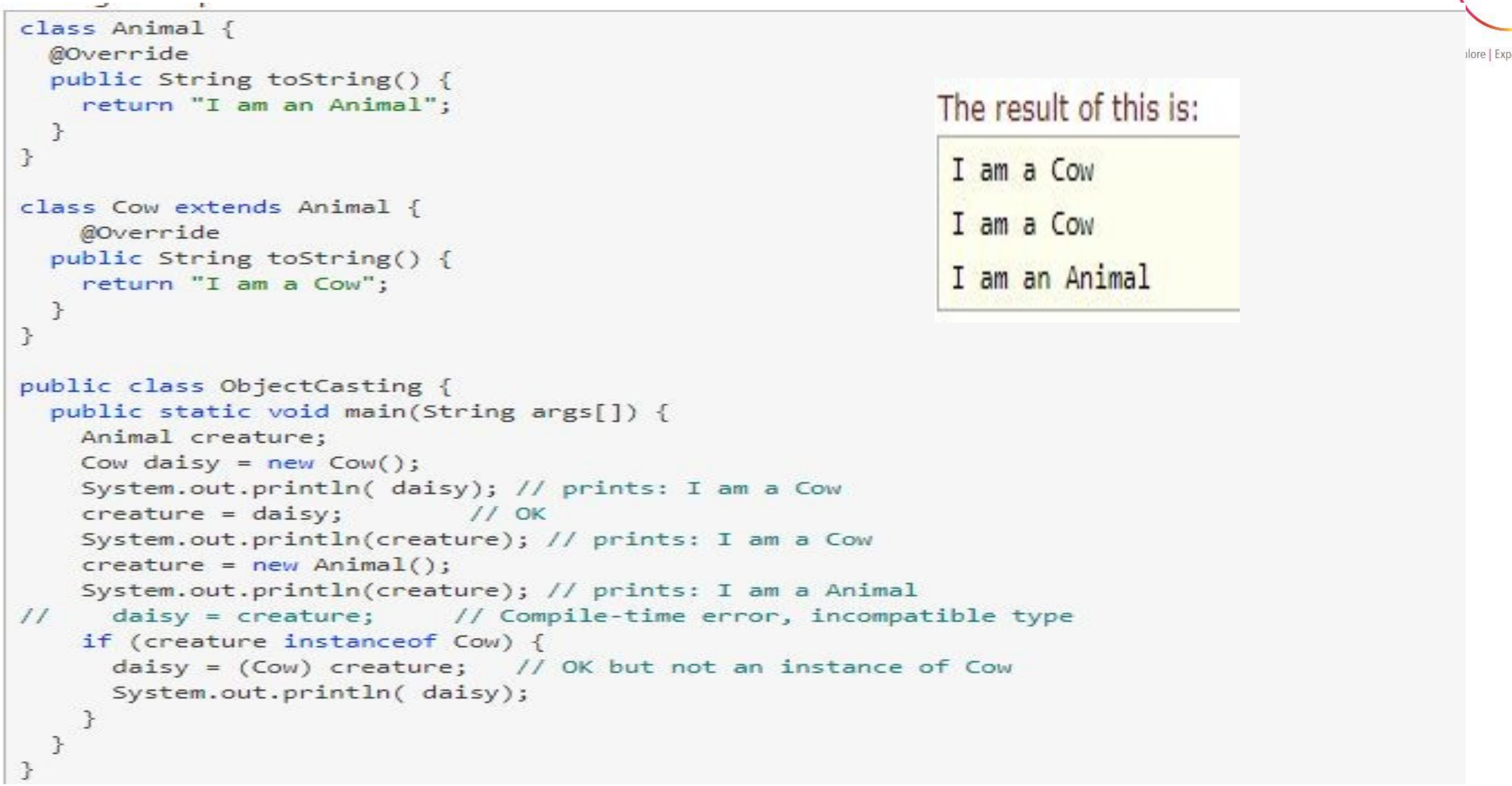


Static Binding and Dynamic Binding

	Static Binding		Dynamic Binding
1	Static Binding is also called as Early binding	1	Dynamic Binding is also called as Late Binding
2	It takes place at Compile-time	2	Binding takes place at the run time
3	Static Binding uses Overloading/ Operator Overloading Method .	3	Dynamic binding uses Overriding Method.
4	Real object is never used in Static Binding.	4	Real object used in the Dynamic Binding.
5	Static Binding can take place using normal functions	5	Dynamic Binding can be achieved using the virtual functions

CASTING OBJECTS

- ❖ A cast, instructs the compiler to change the existing type of an object reference to another type.
- ❖ In Java, all casting will be checked both during compilation and during execution to ensure that they are legitimate.
- ❖ An attempt to cast an object to an incompatible object at runtime will results in a ClassCastException.
- ❖ A cast can be used to narrow or downcast the type of a reference to make it more specific



```
class Animal {
  @Override
  public String toString() {
    return "I am an Animal";
  }
}

class Cow extends Animal {
    @Override
  public String toString() {
    return "I am a Cow";
  }
}

public class ObjectCasting {
  public static void main(String args[]) {
    Animal creature;
    Cow daisy = new Cow();
    System.out.println( daisy); // prints: I am a Cow
    creature = daisy;        // OK
    System.out.println(creature); // prints: I am a Cow
    creature = new Animal();
    System.out.println(creature); // prints: I am a Animal
//    daisy = creature;     // Compile-time error, incompatible type
    if (creature instanceof Cow) {
      daisy = (Cow) creature;   // OK but not an instance of Cow
      System.out.println( daisy);
    }
  }
}
```

The result of this is:

```
I am a Cow
I am a Cow
I am an Animal
```

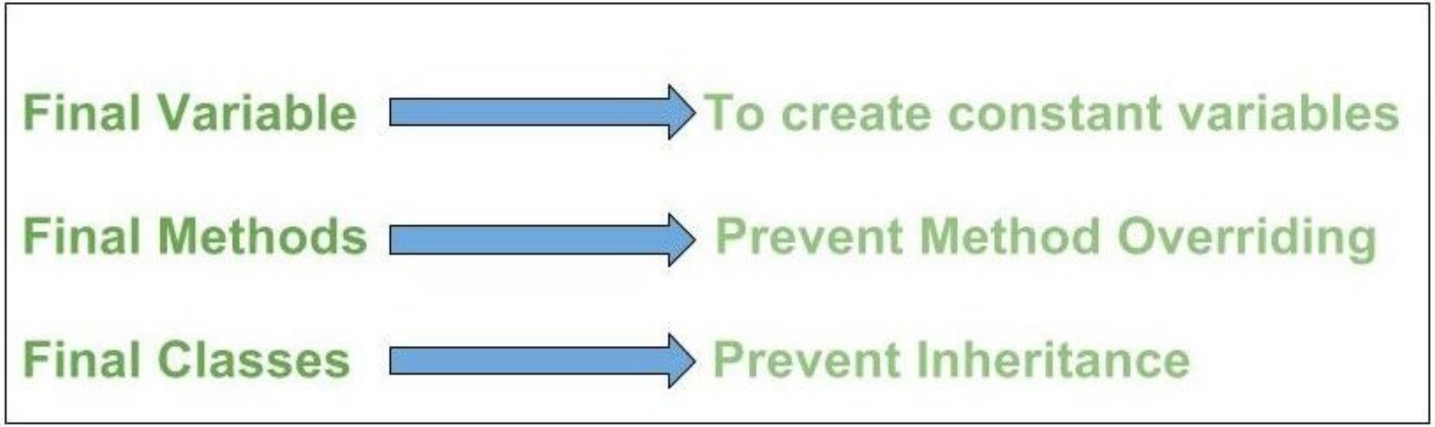
FINAL METHOD AND CLASSES

You can declare some or all of a class's methods *final*. You use the final keyword in a method declaration to indicate that subclasses cannot override the method. The Object class does this—a number of its methods are final.

You might wish to make a method final if it has an implementation that should not be changed and it is critical to the consistent state of the object. For example, you might want to make the getFirstPlayer method in this ChessAlgorithm class final:

```
class ChessAlgorithm {

    enum ChessPlayer { WHITE, BLACK }
    final ChessPlayer getFirstPlayer() {

        return ChessPlayer.WHITE;

    }}
```

- ❖ Methods called from constructors should generally be declared final. If a constructor calls a non-final method, a subclass may redefine that method with surprising or undesirable results.
- ❖ Note that you can also declare an entire class final. A class that is declared final cannot be subclassed. This is particularly useful, for example, when creating an immutable class like the String class.



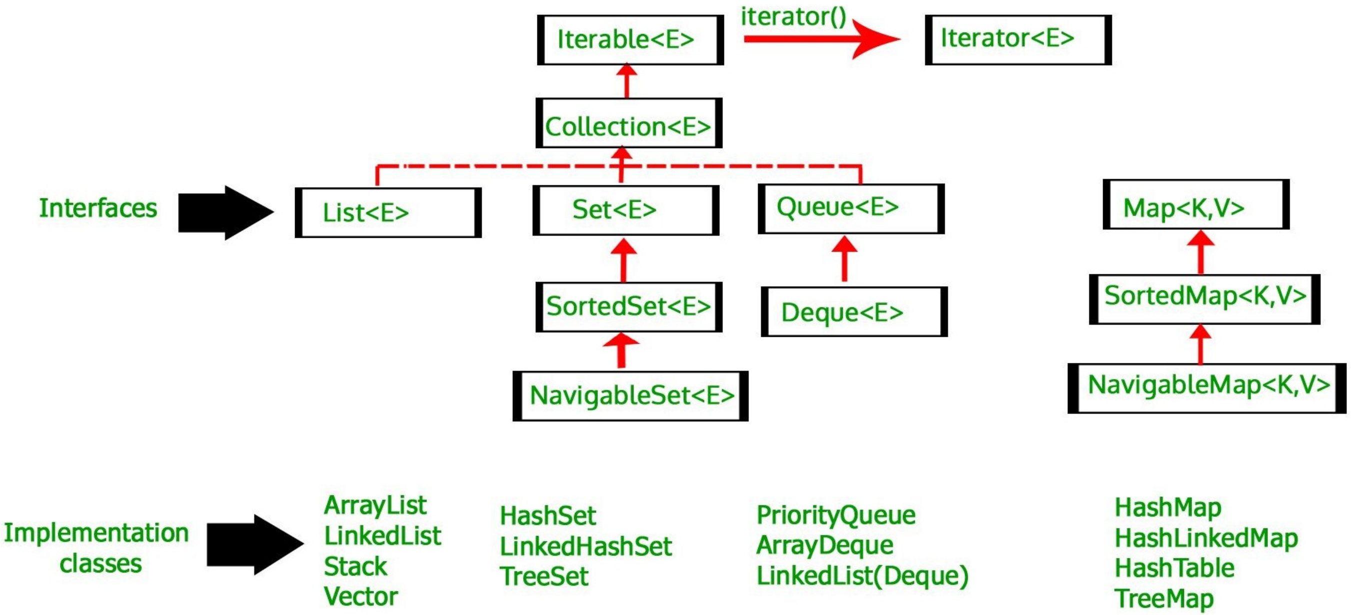
Final Variable → To create constant variables

Final Methods → Prevent Method Overriding

Final Classes → Prevent Inheritance

ARRAYLIST CLASS AND METHODS

- ❖ Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.
- ❖ The important points about Java ArrayList class are:
  - ➢ Java ArrayList class can contain duplicate elements.
  - ➢ Java ArrayList class maintains insertion order.
  - ➢ Java ArrayList class is non synchronized.
  - ➢ Java ArrayList allows random access because array works at the index basis.
  - ➢ In Java ArrayList class, manipulation is slow because a lot of shifting needs to occur if any element is removed from the array list.



Iterable<E>

iterator()

Iterator<E>

Collection<E>

Interfaces

List<E>

Set<E>

Queue<E>

Map<K,V>

SortedSet<E>

Deque<E>

SortedMap<K,V>

NavigableSet<E>

NavigableMap<K,V>

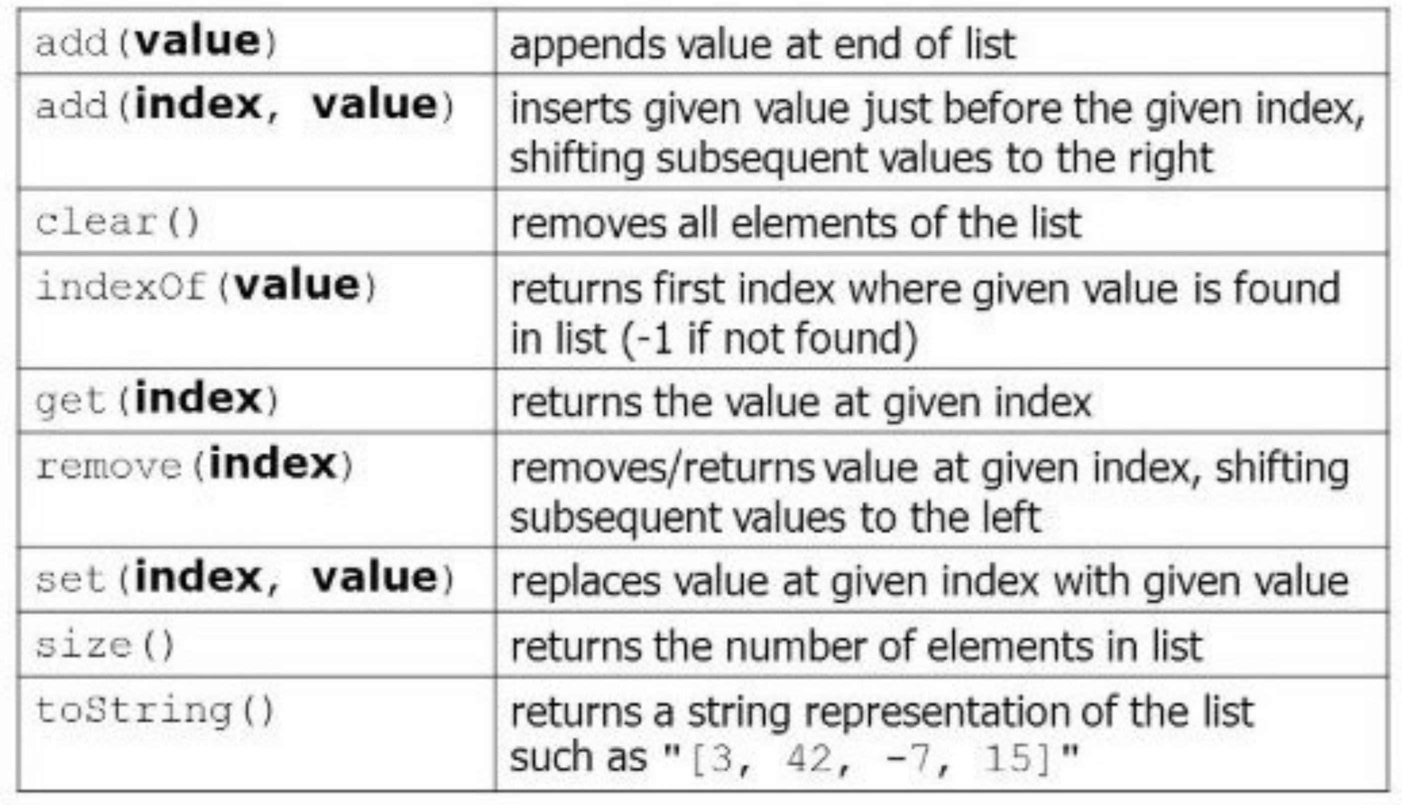
Implementation classes

ArrayList
LinkedList
Stack
Vector

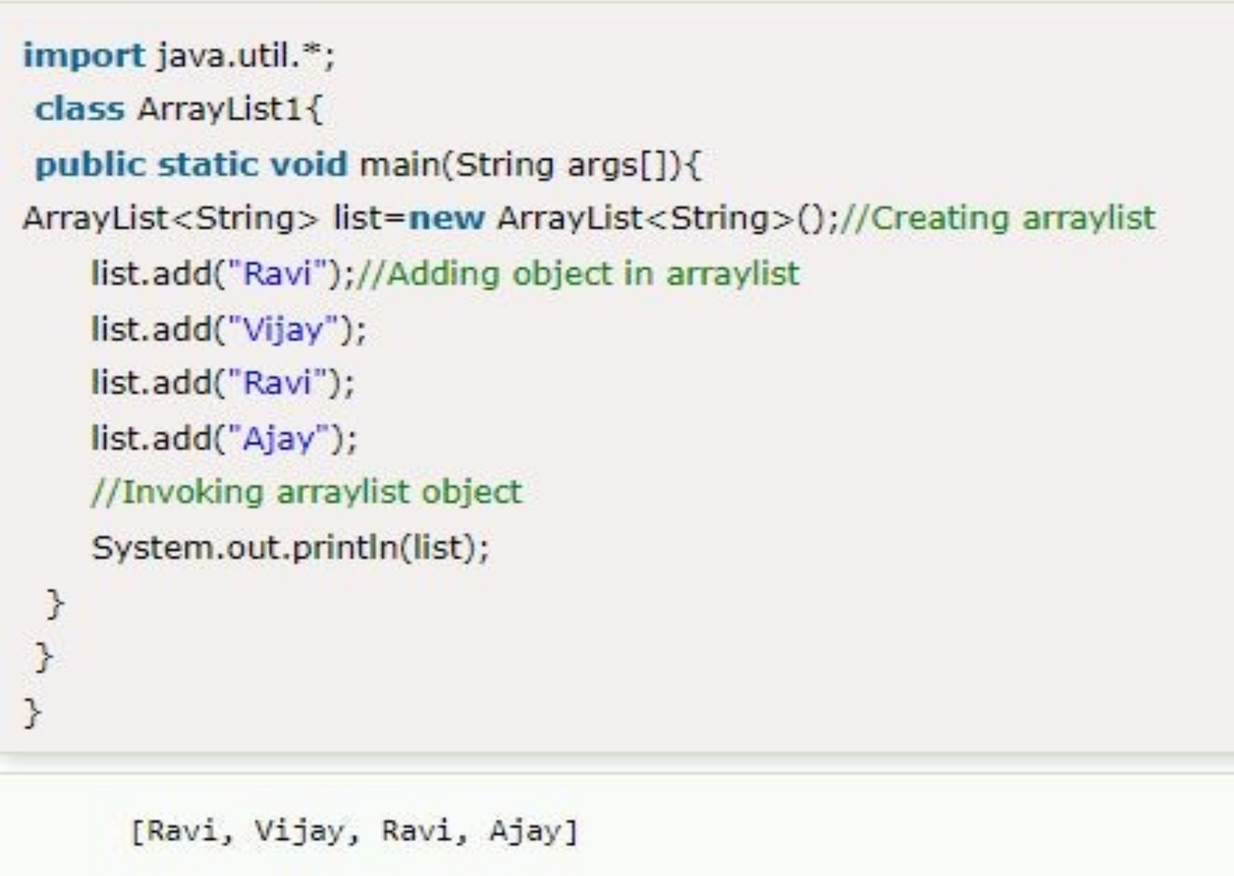
HashSet
LinkedHashSet
TreeSet

PriorityQueue
ArrayDeque
LinkedList(Deque)

HashMap
HashLinkedMap
HashTable
TreeMap



`add(`**value**`)`	appends value at end of list
`add(`**index**`, `**value**`)`	inserts given value just before the given index, shifting subsequent values to the right
`clear()`	removes all elements of the list
`indexOf(`**value**`)`	returns first index where given value is found in list (-1 if not found)
`get(`**index**`)`	returns the value at given index
`remove(`**index**`)`	removes/returns value at given index, shifting subsequent values to the left
`set(`**index**`, `**value**`)`	replaces value at given index with given value
`size()`	returns the number of elements in list
`toString()`	returns a string representation of the list such as "`[3, 42, -7, 15]`"



```
import java.util.*;
 class ArrayList1{
 public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();//Creating arraylist
     list.add("Ravi");//Adding object in arraylist
     list.add("Vijay");
     list.add("Ravi");
     list.add("Ajay");
     //Invoking arraylist object
     System.out.println(list);
  }
 }
}
```

```
[Ravi, Vijay, Ravi, Ajay]
```

POPULAR PROGRAMMING LANGUAGE

Java:

Known for: Being platform-independent, object-oriented, and widely used in enterprise applications, Android development, and big data.
Strengths: Robust, secure, scalable, and has a large community.
Common Uses: Web applications, mobile apps (Android), enterprise software, data analysis, and games.

C:

Known for: Being a low-level language, close to the hardware, and used for systems programming.
Strengths: Fast execution, efficient memory management, and excellent control over hardware.
Common Uses: Operating systems (like Linux and macOS), embedded systems, device drivers, and performance-critical applications.

C++:

Known for: Being a powerful, object-oriented language, often used for high-performance applications.
Strengths: Can be used for both low-level and high-level programming, provides flexibility, and has a large ecosystem.
Common Uses: Games, graphics applications, operating systems, databases, and performance-critical software.

Python:

Known for: Being beginner-friendly, versatile, and popular for data science, machine learning, and web development.
Strengths: Easy to learn, readable code, extensive libraries for various tasks, and a strong community.
Common Uses: Data analysis, machine learning, web development, scripting, and automation.

Choosing the Right Language:

The best language for you depends on your project needs:

Performance-critical applications: C or C++ are often preferred.

Mobile apps: Java (for Android) or Swift (for iOS) are popular choices.

Web development: Python (with frameworks like Django or Flask), Java (with frameworks like Spring), or JavaScript are common.

Data science and machine learning: Python is a dominant language.

https://learn.codemithra.com

THANK YOU