OOP-JAVA
CST 205

Core JAVA Fundamentals

MODULE 2

Author – Milan George Mathew

Syllabus

Primitive Data types - Integers, Floating Point Types, Characters, Boolean. Literals, Type Conversion and Casting, Variables, Arrays, Strings, Vector class.

Operators - Arithmetic Operators, Bitwise Operators, Relational Operators, Boolean Logical Operators, Assignment Operator, Conditional (Ternary) Operator, Operator Precedence.

Control Statements - Selection Statements, Iteration Statements and Jump Statements.

Object Oriented Programming in Java - Class Fundamentals, Declaring Objects, Object Reference, Introduction to Methods, Constructors, this Keyword, Method Overloading, Using Objects as Parameters, Returning Objects, Recursion, Access Control, Static Members, Final Variables, Inner Classes, Command Line Arguments, Variable Length Arguments.

Inheritance - Super Class, Sub Class, The Keyword super, protected Members, Calling Order of Constructors, Method Overriding, the Object class, Abstract Classes and Methods, using final with Inheritance.

Primitive Data Types

Java is a **strongly typed language.** Every variable has a type and every expression has a type and the types are strictly defined. All assignments are checked for **type compatibility** by parameter passing in methods or by explicit type conversions. The Java compiler checks all expressions and parameters to ensure that the types are compatible. Any type mismatches are errors that must be corrected before the compiler will finish compiling the class.

Java has eight primitive data types that are commonly referred to as simple types. They represent single values and not complex objects. They are

- **INTERGERS** – whole value signed numbers
  - byte
  - short
  - int
  - long
- **FLOATING – POINT NUMBERS** – numbers with fractional precision
  - float
  - double
- **CHARACTERS** – symbols in character set, like letters, numbers etc.
  - char
- **BOOLEAN** – true or false
  - boolean

Integers

We have four integer types in java namely, **byte, short, int, long.** They can be positive or negative signed values. Java doesn't support unsigned, positive only integers. In Java, the **width** of an integer data type refers to the **number of bits** it occupies in memory. The width determines the range of values that can be represented by the data type.

Byte

Being the smallest integer type it can have the values from **-128 to 127.** It is useful when dealing with a stream of data, as ASCII values of characters can easily be stored in them. This is signed 8-bit integer where 7 bits are used for integral part and 1 bit for the sign.

```
byte b, c;
```

Short

It is a 16-bit signed integer that can range from **–32,768 to 32,767,** not a commonly used data type. It can be used to store small quantities like dates of a month or year etc.

```
short s, t;
```

Int

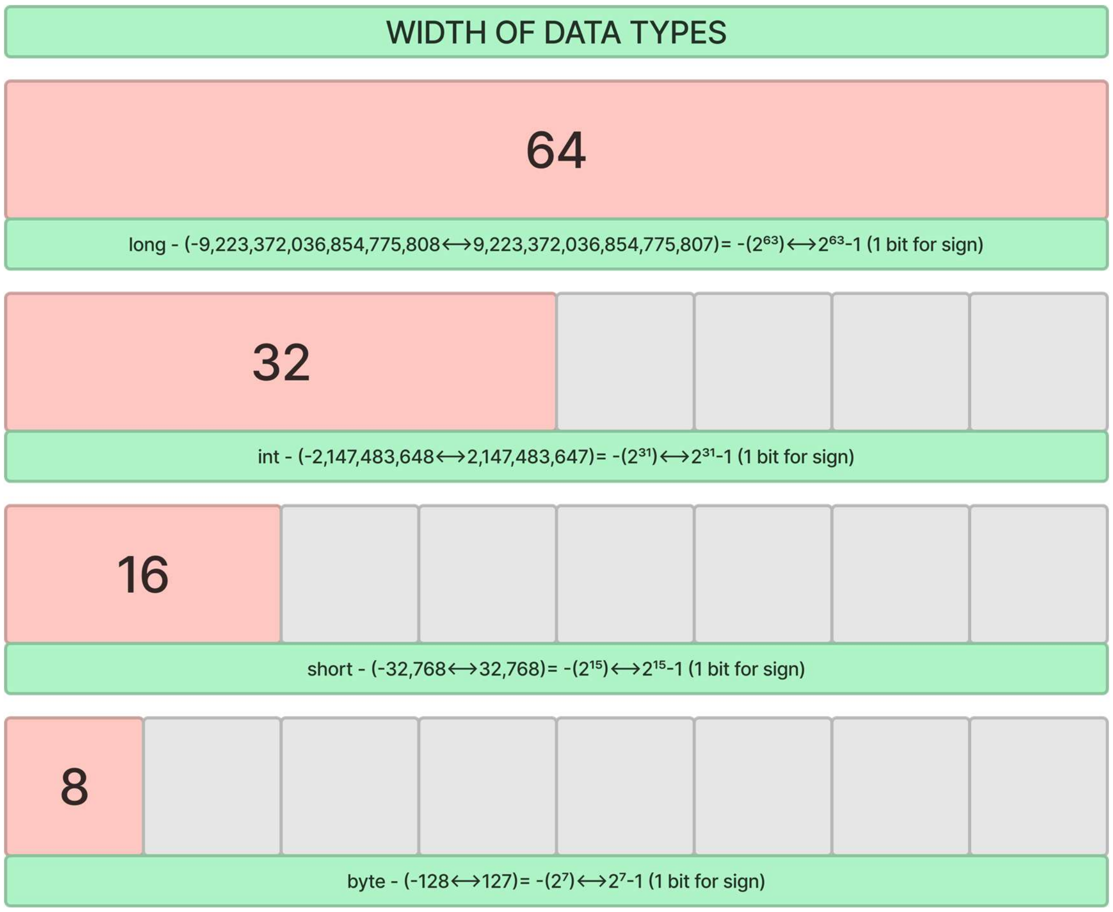
It is the most commonly used integer data type which uses a 32-bit signed format that can range from **–2,147,483,648 to 2,147,483,647.** It can be used for control loops, index arrays and to represent other quantities. When **byte and short** values are used in an expression, they are **promoted to int** when the expression is evaluated.

```
int a, b;
```

Long

A 64-bit signed integer type that is useful for occasion where integer data type is not enough to hold values like timestamps or results of highly complex calculations or large values that are significant. The range varies from **–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807**

```
long l, u;
```


WIDTH OF DATA TYPES
64
long - (-9,223,372,036,854,775,808⟷9,223,372,036,854,775,807)= -(2^{63})⟷2^{63}-1 (1 bit for sign)
32
int - (-2,147,483,648⟷2,147,483,647)= -(2^{31})⟷2^{31}-1 (1 bit for sign)
16
short - (-32,768⟷32,768)= -(2^{15})⟷2^{15}-1 (1 bit for sign)
8
byte - (-128⟷127)= -(2^{7})⟷2^{7}-1 (1 bit for sign)

Floating point types

In Java, **floating-point** types are used to represent **real numbers with fractional parts**. There are two primary floating-point types: **float and double**. These types are used when you need to store numbers that may have a fractional component or require a larger range than integer types can provide. They are used for precise calculations with **fractional precision**.

Float

The **float** data types store **real numbers with fractional part** in a **32-bit format** that is of **single precision**, it is faster for some processors as it takes **half the amount of space** as for double precision. But the value is **imprecise** and the values may be varied by large or small scale. Variables of type float are useful when you need a **fractional component**, but **don't require a large degree of precision**.

It is having an approximate range of: **1.4×10^{-45} to 3.4×10^{38}**

```
float g, u;
```

Double

Double is a **64-bit format** system to represent **double precision** floating point values. Some modern processors work fast on double precision than single precision. Math functions like **sin( ), cos( ), sqrt( )** etc. returns double value.

It is having an approximate range of: **4.9×10^{-324} to 1.8×10^{308}**

```
double dm, cs;
```

Characters

The **char** data type in Java is used to represent a single **16-bit Unicode** character, unlike C/C++ which uses **8-bit width**. It is part of the integral data types in Java and is used to store characters such as letters, digits, and symbols. **Unicode** defines a fully **international character** set that can represent all of the characters found in all human languages. The values ranges from **0 to 65536.** Also there are no negative values to be stored in char.It can be initialized by two ways:

```
char ch1 = '<character>';
char ch2 = < UNICODE of character >;
```

Booleans

The Boolean data type in Java is a primitive data type that represents a binary condition, typically used for expressing **true or false** values. It is the **simplest form** of data type and is commonly used in decision-making and conditional expressions. This is the type , true or false. returned by all relational operators, boolean is also the type required by the conditional expressions that govern the control statements such as if and for.

```
boolean b;
```

Literals

In Java, a literal is a **source code representation of a fixed value**. It is a constant or a fixed value that is **directly written** in the source code of a program. Literals can be used to represent values of primitive data types (such as integers, floating-point numbers, characters, and booleans) as well as strings.

- **Integer Literals**
- **Floating-Point Literals**
- **Boolean Literals**
- **Character Literals**
- **String Literal**

Integer Literals

Any whole number value is an integer literal. There are four bases in which integer literals can be used:

- **Decimal (Base 10)**:
  - Cannot have leading zeros
  - Can use digits from 0 to 9
  - Example: 1,4,35,82
- **Binary (Base 2):**
  - Starts with a prefix **0b or 0B**
  - Can use only digit **0** and **1**
  - Example: `0b10001, 0B1101001`
- **Octal (Base 8):**
  - Starts with a prefix 0
  - Can use only digits from 0 to 7
  - Example: `023, 0342`
- **Hexadecimal (Base 16):**
  - Starts with a prefix **0x or 0X**
  - Can use digits from 0 to 9 and letters A (or a) to F (or f)
  - Example: `0x23f, 0X3ED`

An **Integer Literal** can be **assigned** to a **long type** variable by appending an L (or l) at the end.

```
long longint = 243L;
```

Integer can also be assigned to a char as long as it is within range. Integer literal value is assigned to a byte or short variable as long as it is within range.

From Java 7 and later underscores can be used to improve readability of large numeric literals.

```
int bigNumber = 1_000_000;
```

Floating-Point Literals

Floating-point numbers represent decimal values with a fractional component. There are two notations given as:

- **Standard notation**
    - Consists of a whole number component followed by a decimal point followed by a fractional component.
    - Example: `3.14234, 9.84`
- **Scientific notation**
    - Consist of a standard notation part followed by an exponent component denoted after E (or e) showing the number of 10s to be multiplied.
    - Example: `6.022E23, 314159E-05`

Floating-point literals in Java are **double precision** by default. To specify a float literal, we must append an F (or f) to the constant. We can also explicitly specify a double literal by appending a D or d.

Boolean Literals

Boolean literals are the simplest of all literals. There are only two logical values that takes the values **true and false**. The values of true and false **do not convert into any numerical representation**. Which means, true is not equal to 1 and false is not equal to 0 in Java.

Character Literals

Characters in Java are indices into the **Unicode** character set. They are **16-bit** values that can be converted into integers and manipulated with the integer operators, such as the addition and subtraction operators. A literal character is represented inside a pair of single quotes. All of the visible **ASCII** characters can be directly entered inside the quotes. **Escape character sequences** are used to print non-printable characters like newline, tab space etc. Some are given below:

- '\n' - Newline
- '\\'' – Single Quote
- '\141' – octal digit representation of a character in Unicode
- '\u0061' – hexadecimal digit representation of a character ins Unicode
- '\\\\' – backslash
- '\t' – tab

String Literals

String literals in Java are specified like they are in most other languages by enclosing a sequence of characters between a pair of **double quotes**.

Variables

In Java, a variable is a named storage location that holds a value, which can be changed during the execution of a program. Variables are used to store and manipulate data in a Java program. When you define a variable, you are essentially creating a named memory location with a specific data type to hold values of that type. It is the basic unit of storage. Its basic syntax of variable declaration and initialization is given by:

```
dataType variableName;

dataType variableName = initialValue;
```

Variables are confined to the scope they are defined to, which defines its visibility and lifetime.

Declaring a Variable

All variables must be declared before they can be used. The type defines the data type of the value that the variable may hold and the identifier is the name of the variable that are not supposed to be one of the keywords and has to follow rules to define identifiers.

Dynamic Initialisation

Java allows variable to be initialized dynamically using an expression that is valid at the time of declaration. For example

```
double a = 3.0, b = 4.0;

double c = Math.sqrt(a * a + b * b);
```

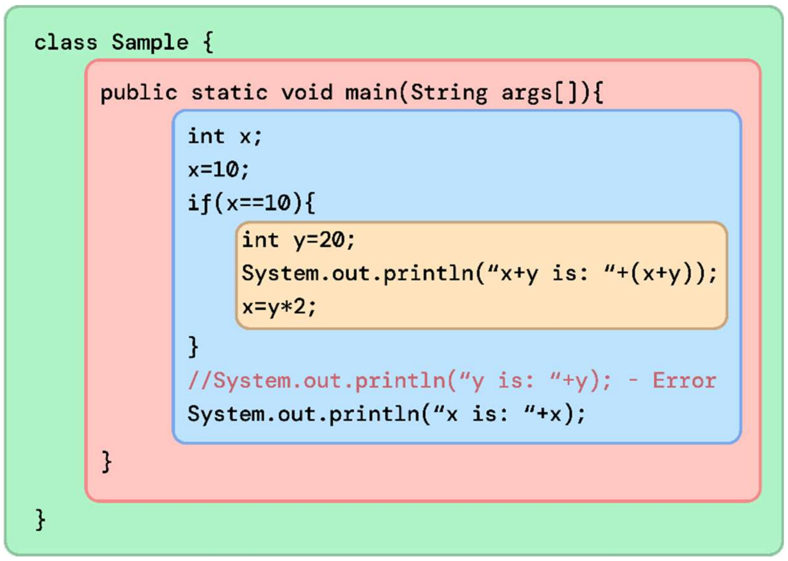
Scope and Lifetime of variables

Java allows variables to be declared within any block, which defines a scope. A block begins with an opening curly brace and ended by a closing curly brace. A scope determines what objects are visible to other parts of your program. Scope also determines the lifetime of those objects.

Two major scopes in Java are:

- Scope defined by a **class**
- Scope defined by a **method**

Variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope. Scopes can be nested. Each time you create a block of code, we are creating a new, nested scope. The outer scope encloses the inner scope. This means that objects declared in the outer scope will be visible to code within the inner scope.



```
class Sample {
    public static void main(String args[]){
        int x;
        x=10;
        if(x==10){
            int y=20;
            System.out.println("x+y is: "+(x+y));
            x=y*2;
        }
        //System.out.println("y is: "+y); - Error
        System.out.println("x is: "+x);
    }
}
```

Variables are **created** when their **scope is entered**, and **destroyed** when their **scope is left**. This means that a variable will not hold its value once it has **gone out of scope**. Variable can be **reinitialized** each time it enters the block in which it is declared. Although blocks can be nested, you **cannot declare** a variable to have the **same name** as one in an **outer scope**.

Type Conversion and Casting

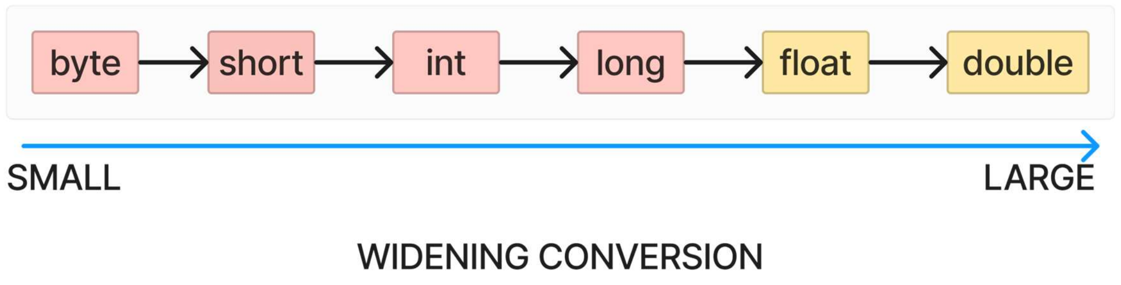
If the two types are **compatible**, then Java will perform the conversion **automatically**, this is called **implicit type conversion or implicit type casting**. The conversion between incompatible types is to be done **explicitly.**

Implicit Conversion

When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:

- The types are compatible
- The destination is equal or larger than source type

This is also termed as widening as it happens from a small data type to a larger one thus widening the memory allocation for the variable. For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. No automatic conversions from the numeric types to char or boolean. Java also performs an automatic type conversion when a literal integer constant is stored into variables of type byte, short, long, or char.



Explicit Conversion

To convert a data type to a one that is smaller in size or is incompatible is done through casting or explicit type conversion. It is generally of the form

```
(target-type) value
```

If the value of the converted type is larger than the size of target type then it will be reduced to modulo by the target values range. For example consider

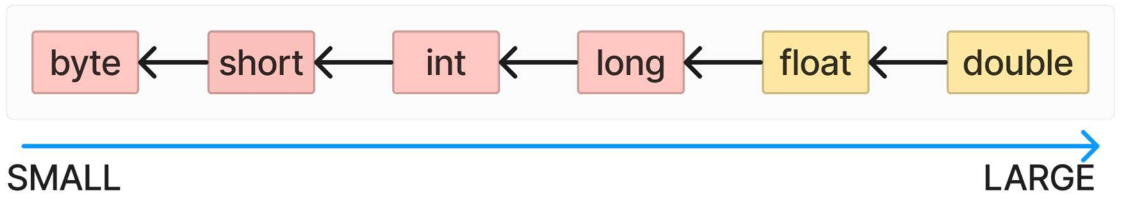
```
int a=300;
byte b=(byte) a;
```

if a is greater than 256 (2^8) which is the maximum size of byte datatype then b= 300%256=44.

While a **float** is converted to **int** then a different type of truncation is done called truncation, which is to truncate the decimal values.

```
float a=353.23;
int b=(int) a;
```

the value of b will be 353 in this case



NARROWING CONVERSION

Automatic conversions

Automatic promotion refers to the implicit conversion of smaller data types to larger data types in expressions or assignments. Automatic promotion ensures that smaller data types are widened to larger ones to avoid data loss and maintain precision in expressions involving mixed data types.

For example, take:

```
byte a = 40, b = 50, c = 100;
int d = a * b/ c;
```

The data types of a, b, c are in bytes but when the result is computed to an integer variable d the conversion is done automatically

Type promotion rules

- All **byte, short and char** are promoted to **int**
- If **one** operand is a **long**, the whole expression is promoted to **long**.
- If **one** operand is a **float**, the entire expression is promoted to **float**.
- If **any** of the operands is **double**, the result is **double**.

Arrays

An array is a data structure that allows you to store multiple values of the same type under a single variable name. Arrays are used to group together elements of the same data type, and each element in the array can be accessed using an index. It can be made of any type and can be of more than one dimension and the elements are indexed, so that accessing of elements can be done using indices starting from 0

One dimensional array

A one-dimensional array in Java is a linear collection of elements of the same data type, arranged in a single row. Each element in the array is identified by its index, starting from 0 for the first element. There are two syntaxes used to declare a one-dimensional array

```
type varname[];
type[] varname;
```

This declaration just declares that a varname is an array and does not provide allocated space for the array in memory. In Java all arrays are dynamically allocated. To allocate space we use the new operator and assign it to the variable:

```
varname[] = new type[size]
```

the size refers to the number of elements it can hold. This two-step process of declaration and memory allocation can be done easily by:

```
type varname[] = new type[size]
```

The values of elements can be added using the index and assigning values to the index position.

Arrays can be initialized (give values) when they are declared. An array initializer is a list of comma-separated expressions surrounded by curly braces. In this syntax the new operator is not needed. For example:

```
int a = {1,3,5,7,9,0};
```

If you try to access elements outside the range of the array (negative numbers or numbers greater than the length of the array), it will cause a **run-time error**. This exception is given as ArrayIndexOutOfBoundsException.

Multidimensional array

They are referred to as array of arrays. The declaration of a multidimensional array is done by adding extra square brackets to the usual declaration. For example:

```
type varname[][];
type[][] varname;
```

this declares a 2-dimensional array.

```
int b[][]= new int[4][5];
```

declares a 2D array and allocates memory as if it has 4 rows and 5 columns or we can store four 5 membered arrays in b. While allocating memory we need only the first dimension to be declared the subsequent dimension can be varied according to the users' choice, for example:

```
int a[][] = new int[2][];

a[0] = new int[3];

a[1] = new int[2];
```

this code snippet declares and allocates the memory for an array which can have two arrays which are of size 3 and 2 respectively. We can initialize the values into a multi-dimensional array by using the curly bracket notation for every array including both inner and outer arrays. For example:

```
int a[][] = {{1,2,3}, {3,4,5}} ;
```

String class

The String class in Java is a fundamental class that belongs to the **java.lang** package. It is widely used to represent sequences of characters (text) and provides numerous methods for manipulating and working with strings. Every string is an object of the String class, and is assigned by a quoted string constant. A String object can be assigned to another String object.

```
String str = "Test String";

System.out.println(str);
```

An array of characters works same as Java string.

```
char[] ch = {'H','e','l','l','o'};

String s = new String(ch);
```

This snippet will be same as:

```
String s = "Hello";
```

String Methods

length()

The length of a string can be found with the **length**() method.

toUpperCase() and toLowerCase()

To convert from lower case to upper case and upper case to lower case respectively.

indexOf()

The indexOf() method returns the index (the position) of the first occurrence of a character or specified text in a string (including whitespace).

Concatenation

There are two ways to concatenate strings

- Using the '+' operator
- Using **concat**() method

If we add a number and a string, the result will be a string concatenation

Vector Class

The Vector class is a part of the Java Collections Framework and is located in the java.util package. It implements a dynamic array that can grow or shrink in size, allowing for the storage of elements in a sequence. The Vector class in Java is a resizable array implementation of the List interface. It is synchronized, meaning it is thread-safe, and it provides methods to add, remove, and access elements in a flexible and dynamic manner. Vector is synchronized, and it contains many legacy methods that are not part of the Collections Framework. All vectors start with an initial capacity (size) which gets expanded if more than the specified size of elements is added and also makes space for additional objects.

The amount of extra space allocated during each reallocation is determined by the increment that you specify when you create the vector. If we don 't specify an increment, the vector's size is doubled by each allocation cycle.

```
class Vector<E>
```

some of its constructors are:

```
Vector( )

Vector(int size)

Vector(int size, int incr)

Vector(Collection<? extends E> c)
```

- **size** – defines the initial capacity which by default is 10
- **incr** – specified the number by which the size should be incremented
- **c** – a collection that is used to create a vector.

Vector has some protected data members given as:

```
int capacityIncrement;
```

Stores the increment value.

```
int elementCount;
```

Stores number of current elements

```
Object[ ] elementData;
```

The array that holds the vector is stored in elementData.

Operators

There are different types of operators in Java:

- **Arithmetic Operators**
- **Bitwise Operators**
- **Relational Operators**
- **Boolean Logical Operators**
- **Assignment Operators**
- **Ternary Operator**

Arithmetic operators

The basic arithmetic operations addition, subtraction, multiplication, and division works for all numeric types.

+	→	addition
-	→	subtraction
*	→	multiplication
/	→	division

Subtraction operator also acts as a unary operator to convert the value to signed number. **Modulus operator** is a special operator which return the remainder on division given by (%). I can be applied on integer as well as floating point literals.

Arithmetic operators can further be extended as **Arithmetic Compound Assignment operators:**

```
Variable (operator)= expression;
```

This is the general form of an arithmetic compound assignment operator.

a += b	→	a = a + b
a -= b	→	a = a - b
a *= b	→	a = a * b
a /= b	→	a = a / b
a %= b	→	a = a % b

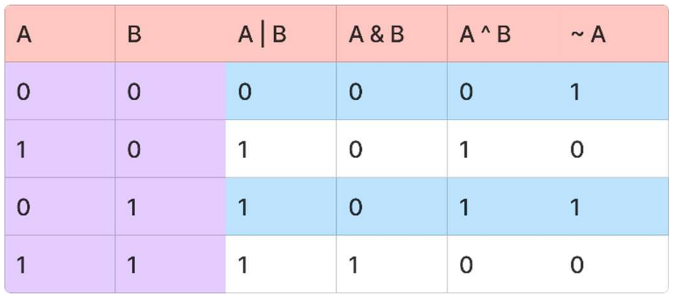
Other of two operators that perform arithmetic are increment operators:

- **Pre-increment** – ++x → increments value before operation
- **Post-increment** – x++ → increments value after operation

Bitwise operator

There are many bitwise operators given as:

- Logical
  - Basic
    - ~ → Bitwise NOT
    - & → Bitwise AND
    - | → Bitwise OR
    - ^ → Bitwise XOR
  - Compound Assignment
    - &= → Bitwise AND assignment
    - |= → Bitwise OR assignment
    - ^= → Bitwise XOR assignment
- Shift operators
  - Basic
    - >> → shift right
    - >>> → shift right zero fill
    - << → shift left
  - Compound Assignment
    - >>= → shift right assignment
    - >>>= → shift right zero fill assignment
    - <<= → shift left assignment



A	B	A | B	A & B	A ^ B	~ A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

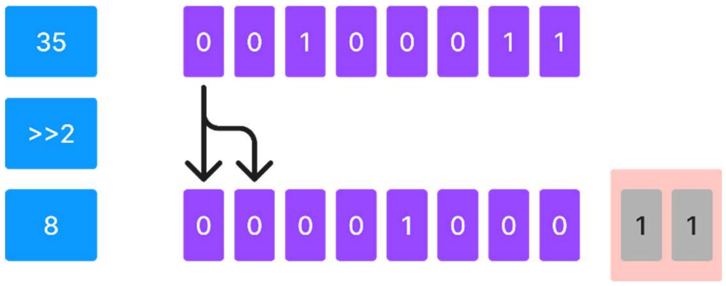
Right Shift

Each time you shift a value to the right, it divides that value by two—and discards any remainder. When you are shifting right, the top (leftmost) bits exposed by the right shift are filled in with the previous contents of the top bit. This is called sign extension and serves to preserve the sign of negative numbers when you shift them right. For example

```
int a = 35;
```

```
a = a>>2;
```

The result would be 8



Unsigned right shift operator

It is just the same as right shift except that it will fill o to the binary that got shifted from left so that the sign is not a matter anymore.

Relational Operators

The operators used to compare values of literals are known as operators

-	==	→	Equal to
-	!=	→	Not equal to
-	>	→	Greater than
-	<	→	Less than
-	>=	→	Greater than or equal to
-	<=	→	Less than or equal to

Boolean Logical Operators

They are used to compare Boolean states:

-	&	→	Logical AND
-	|	→	Logical OR
-	^	→	Logical XOR
-	||	→	Short Circuit OR
-	&&	→	Short Circuit AND
-	!	→	Logical Unary NOT
-	&=	→	AND assignment
-	|=	→	OR assignment
-	^=	→	XOR assignment
-	==	→	Equal to
-	!=	→	Not Equal to

Short Circuited Logical Operators

They are the secondary versions of Boolean AND and OR operators. In general, an OR returns true for the first expression to be true even if the second is false or true, an AND returns false for the first value to be false even if the second is true or false. Thus, in both of these cases the second expression is not considered and thus the outcome is just determined by the left operand and thus it shortens the process and so it is called Short Circuited Logical Operators.

-	||	→	gives true if first operand true else check for the second operand
-	&&	→	gives false if first operand is false else check for the second operand

Assignment Operator

The assignment operator is given by "=" and this can be used to assign multiple values.

```
x = y = z = 100;
```

sets 100 as the value for all of the variables

Ternary Operator

It is a shorthand operation for a single if-else statement, it has the general form:

```
expression1 ? expression2 : expression3
```

- **expression1** – the expression to be evaluated to return a true or false
- **expression2** – if the expression1 is true then this expression is returned
- **expression3** – if the expression1 is false then this expression is returned

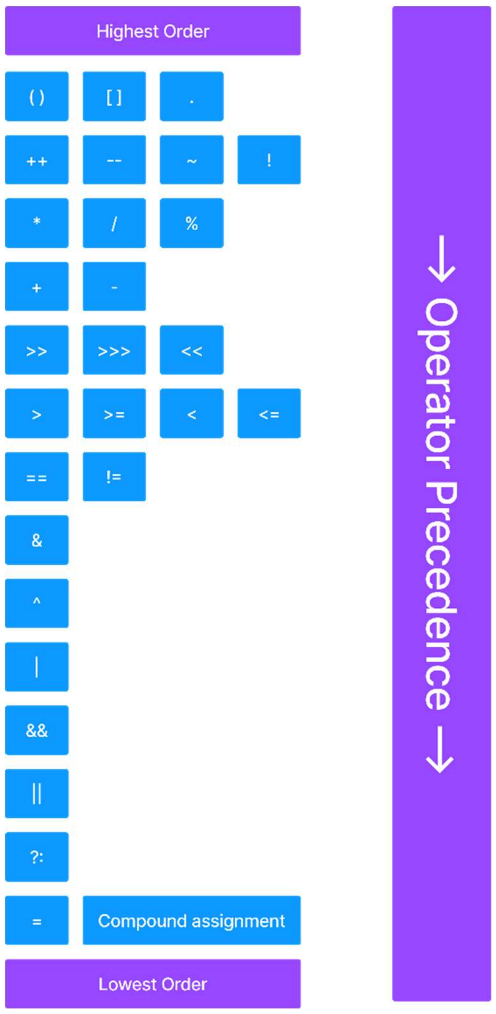
Both expression2 and expression3 are required to return the **same type**, which **can't be void.**

For example:

```
int a=3,b=5;

int c=(a>b?a:b);
```

will assign b to c as the expresiion a>b is false.



Operator Precedence

Operator precedence in Java determines the order in which operators are evaluated when an expression contains multiple operators. Operators with higher precedence are evaluated first. If operators have the same precedence, their associativity (whether they are left-associative or right-associative) comes into play.
The Right to Left Associative operators are

- Unary Operators
- Assignment Operators
- Conditional Operators

All others are Left to Right Associative. This suggest the direction in which the expression is to be evaluated if there are operators of same precedence

Control Statements

Control statements in programming languages are constructs that alter the flow of program execution based on certain conditions or loops. They help in making decisions, repeating code, and creating structured programs. Control statements allow programmers to create flexible and responsive programs by controlling the flow of execution based on conditions and iterations. They are essential for writing structured and efficient code. There are three categories of control statements:

- **Selection Statements** – it allows the program to select the flow of control based on a condition
- **Iteration Statements** – it allows the program to repeat one or more statements till a condition is satisfied
- **Jump Statements** – it allows the program to act in a non linear control flow

Selection Statements

The Selection Statements also called decision making statements are those control the flow of a program's execution based up on conditions that depend of runtime factors. Java supports two selection statements:

- if
- switch

if Statement

if statement is Java's conditional branch statement. It can be used to route program execution through different paths. Syntax of a simple if statement is given by

```
if (condition){

      // code to be executed if true

}
```

if-else Statement

They are used to direct the program to another block of code that can execute if the given condition was false. Generally given as:

```
if (condition) statement-block1;

else statement-block2;
```

Statement may be a single statement or a compound statement enclosed in curly

Nested if

An if statement inside another if or else statement is collectively called nested ifs they are used to check for more conditions that has already passed or failed another condition which was checked for in prior.

```
if (condition1){
      if(condition2){
            // block to be executed if both conditions are passed
      }
      else{
            // block to be executed if condtion1 was true and 
condition 2 was false
      }
}
else{
      // block of code to be executed if condtion1 is false
}
```

if-else if Ladder

The "if-else if ladder" is a structure in Java used when there are multiple conditions to be checked in a sequential manner. It consists of a series of **if** and **else if** statements, and the associated code block of the first true condition is executed. If none of the conditions is true, the else block (if present) is executed, that is in an top down flow As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. The last else acts as a default condition; that is, if all other conditional tests fail, then the last else statement is performed.

```
if (condition1) {
    // code to be executed if condition1 is true
} else if (condition2) {
    // code to be executed if condition2 is true
} else if (condition3) {
    // code to be executed if condition3 is true
} else {
    // code to be executed if none of the conditions is true
}
```

Switch Statement

The switch statement is Java's multiway branch statement. It is a better alternative than a large series of if-else-if statements. Each of the values specified in the case statements must be of a type compatible with the expression.

```
switch (expression) {

    case value1:

        // code to be executed if expression equals value1

        break;

    case value2:

        // code to be executed if expression equals value2

        break;

    // additional cases as needed

    default:

        // code to be executed if expression doesn't match any case

}
```

Default statement is optional. The break statement is used inside the switch to terminate a statement sequence. When a break statement is encountered, execution branches to the first line of code after the entire switch statement. This has the effect of "jumping out" of the switch.

Switch statements can also be nested like if statements.

The switch differs from the if in that switch can only test for equality, whereas if can evaluate any type of Boolean expression. No two case constants in the same switch can have identical values, but a switch statement and an enclosing outer switch can have case constants in common. A switch statement is usually more efficient than a set of nested ifs.

Features

- When Java compiler compiles a switch statement, it will inspect each of the case constants and create a "jump table" that it will use for selecting the path of execution depending on the value of the expression.
- Thus, a switch statement will run much faster than the equivalent logic coded using a sequence of if-else statements.
- The compiler can do this because it knows that the case constants are all the same type and simply must be compared for equality with the switch expression. The compiler has no such knowledge of a long list of if expressions.

Iteration Statements

Iteration statements in Java are used to repeatedly execute a block of code as long as a certain condition is true or for a fixed number of times. The three primary iterative statements in Java are:

- for
- while
- do-while

they can also be termed as looping statements.

while loop

The while loop is Java's most fundamental loop statement. It is an ENTRY CONTROLLED loop as the while loop continues to allow the flow of control to enter the code block until the given condition is false.

```
while (condition) {

    // code to be repeated while the condition is true

}
```

- condition → true : the block continues to execute
- condition → false : the execution of the while loop ends and the flow doesn't enter the block.

The body of the while (or any other of Java's loops) can be empty. This is because a null statement is syntactically valid in Java.

do-while loop

The do-while loop in Java is a control flow statement that is similar to the while loop. However, in a do-while loop, the condition is checked after the execution of the loop body. This means that the code inside the loop is guaranteed to be executed at least once, regardless of whether the condition is initially true or false.

```
do {

    // code to be repeated at least once and then while the
condition is true

} while (condition);
```

do-while is an EXIT CONTROLLED loop as the block is exited first then checked for the condition again and then continues if true.

Providing a true value in the condition can make both of these while and do-while loops infinite

for loop

The for loop in Java is a control flow statement that allows you to repeatedly execute a block of code based on a loop control variable. It is often used when the number of iterations is known in advance.

```
for (initialization; condition; update) {

    // code to be repeated while the condition is true

}
```

- **initialization** – initializes the control variable to an initial value
- **condition** – defines the condition to be checked
- **update** – defines how the control variable is to be updated like an increment or decrement

When the loop first starts, the initialization portion of the loop is executed. It acts as a loop control variable (counter). The initialization expression is only executed once. After initialization the condition is evaluated and if it is true then the inside block of code executes and exits at exit the control variable is updated and then it is again checked with the condition and continues till the updated variable of any of the subsequent updates becomes false with respect to the condition.

```
for ( ; ; ) {

      // code block

}
```

Makes an infinite loop

For loop has a special version given as for-each version. It is used to iterate through collections and arrays till there are no elements in them. The generally syntax is:

```
for(type var : collection) {

      // code block

}
```

Loops can be nested to have and execution continue for a defined number of times till the outer control variable updates. Nesting of loops are done by creating new looping scopes inside another loop

Jump Statements

Jump statements in Java are used to control the flow of program execution by transferring control to a specific point within the code. Java supports three main jump statements:

- break
- continue
- return

break statement

The break statement serves three uses:

- terminating a switch case statement sequence
- exit a loop
- acts like a civilized form of goto statement

break as a form of goto

By using this form of break, you can, for example, break out of one or more blocks of code. We can label the scopes or block of codes then using it with the label which states to break out the control from the given block. For example:

```
first: {
        second: {
                third: { if(true) break second; }
        }
}
```

Here the blocks have been labelled by names first second and third. When the break statement executes it breaks out from both the third and second code block and give control to the first block.

continue statement

The continue statement in Java is used within loops to skip the rest of the code inside the loop for the current iteration and move on to the next iteration. It is often used to avoid executing certain code based on a specific condition. In a while and do-while loops, a continue statement causes control to be transferred directly to the conditional expression that controls the loop. In a for loop, control goes first to the iteration portion of the for statement and then to the conditional expression.

return statement

The return statement in Java is used to exit a method and, optionally, return a value to the calling code. When a return statement is encountered, the control of the program is transferred back to the caller, and the value specified in the return statement (if any) is sent back.

Object Oriented Programming in Java

OOP is a powerful paradigm that transforms the way we think about and structure our code. At its core, OOP is about modeling the real world as a collection of objects, each with its own state and behavior. Java, a versatile and widely-used programming language, fully embraces the principles of OOP.

Class Fundamentals

In Java, a class is a fundamental building block that encapsulates data and behavior into a single unit. Classes serve as **blueprints** or **templates for creating objects**, and they provide a way to structure and organize code in an object-oriented manner. An object is an instance of a class. A class creates a logical framework that defines the relationship between its members. A class defines a new type of data.

Class Declaration

To declare a class, you use the class keyword, followed by the class name. The body of the class is enclosed within curly braces { }. Classes can be declared with various access specifiers depending on the preferred scope of the class.

Class Definition

While defining a class it is necessary to define the instance variables and methods that are to be the members of any object created out of the class. The data or variables, defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. The data for each object is separate and are often unique. Functions inside a class are called methods and collectively the instance variable and methods are called members of the class.

The general form of a class definition is given as:

```
class classname {

      type instance_var_1;

      type instance_var_2;

      …

      type instance_var_N;

      type methodname_1 ( { parameter list } ) {

            // body of method 1

      }

      type methodname_2 ( { parameter list } ) {

            // body of method 2

      }

      …

      type methodname_N ( { parameter list } ) {

            // body of method N

      }

}
```

Object Declaration

When we create a class, we are creating a new data type. We can use this type to declare objects of that type. Creating an object of a class is done by

- firstly, declaring a variable of the class type
- secondly creating an object with **new** operator and assigning to the variable

The variable first created can be done by the declarative statement:

```
className objectName;
```

This variable is used to store the object created using the new operator:

```
objectName = new className();
```

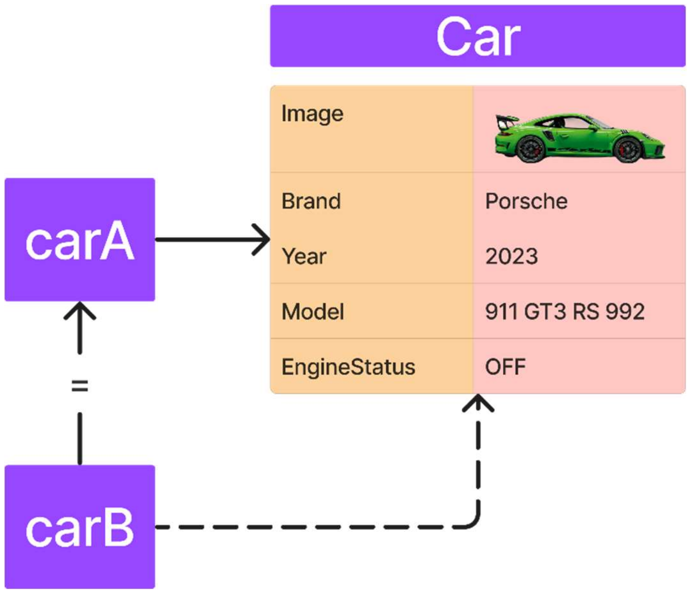
This two step process can be done by just a single line of code by

```
className objectName = new
        className();
```

An object of a class can be assigned to another variable of the same type to create a new object variable for the same object. For example consider a class Car and two variable of type Car such that one is made an instance and other was just a declaration:

```
Car carA = new Car();

Car carB = carA;
```



Class	Object
Template for creating objects	Instance of class
Logical Entity	Physical Entity
Declared using class keyword	Created using new operator
Class does not get any memory when created	Object gets memory when created by new
Class is declared only once	Many objects can be created

Methods

Methods are blocks of code that perform a specific task or action. They are defined within classes and provide a way to encapsulate behaviour.

- Methods facilitate code reusability by allowing you to define behaviour that can be called from different parts of your program.
- Methods can have parameters to receive input values and a return type to send back a result.
- The **void** return type is used when a method does not return a value.
- The **return** statement is used to exit a method and optionally return a value.
- Methods can be called on objects (instances of a class) or directly if they are declared as **static**.
- Java supports method overloading, allowing multiple methods with the same name but different parameter lists.

The general form of a method is given as:

```
type methodName( { parameterList } ) {
      //body of method
}
```

We can use an example to understand all of these concept:

A class Car can be depicted as in the figure

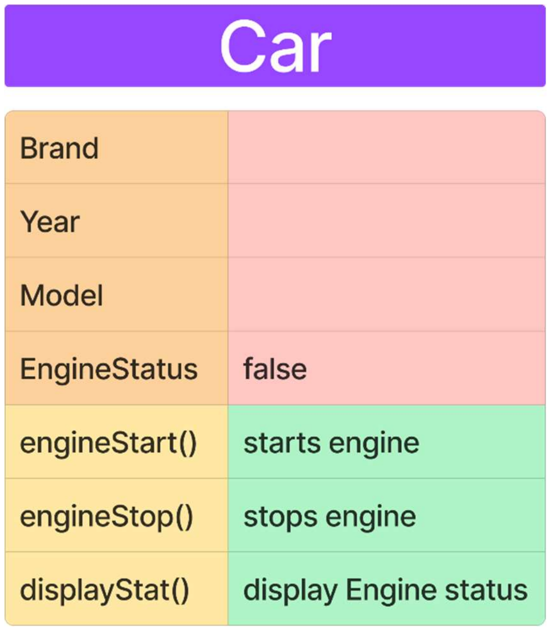
There are instance variables that are:

- Brand
- Year
- Model
- EngineStatus

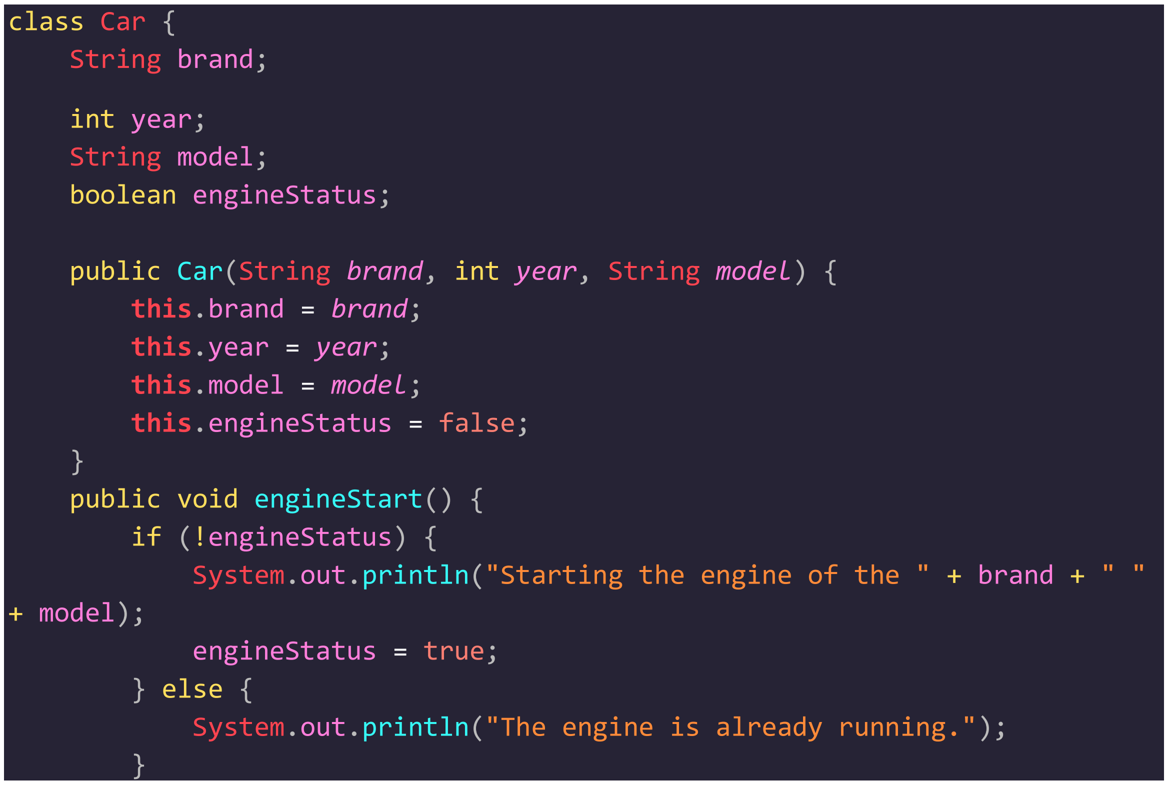
There are also methods defined in the class:

- engineStart() – to start engine
- engineStop() – to stop engine
- displayStats() – to show engine status

The java class representation can be given as follows



Car	
Brand	
Year	
Model	
EngineStatus	false
engineStart()	starts engine
engineStop()	stops engine
displayStat()	display Engine status



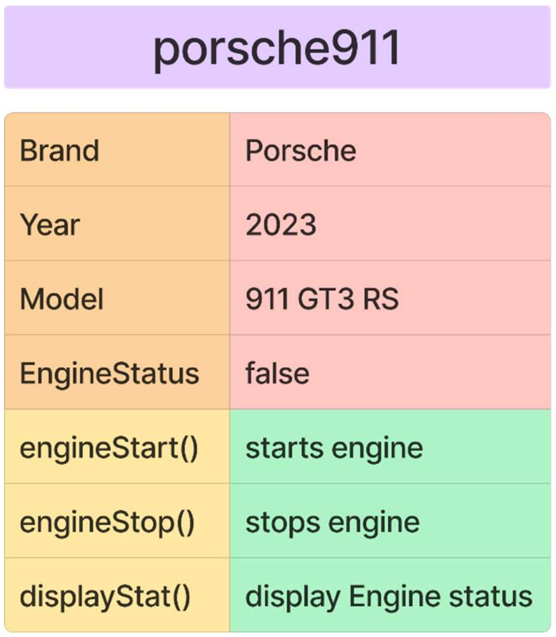
```
class Car {
    String brand;

    int year;
    String model;
    boolean engineStatus;

    public Car(String brand, int year, String model) {
        this.brand = brand;
        this.year = year;
        this.model = model;
        this.engineStatus = false;
    }
    public void engineStart() {
        if (!engineStatus) {
            System.out.println("Starting the engine of the " + brand + " "
+ model);
            engineStatus = true;
        } else {
            System.out.println("The engine is already running.");
        }
    }
    public void engineStop() {
        if (engineStatus) {
            System.out.println("Stopping the engine of the " + brand + " "
+ model);
            engineStatus = false;
        } else {
            System.out.println("The engine is already stopped.");
        }
    }
    public void displayStats() {
        System.out.println("Engine Status: " + (engineStatus ? "Running" :
"Stopped"));
    }
}
```

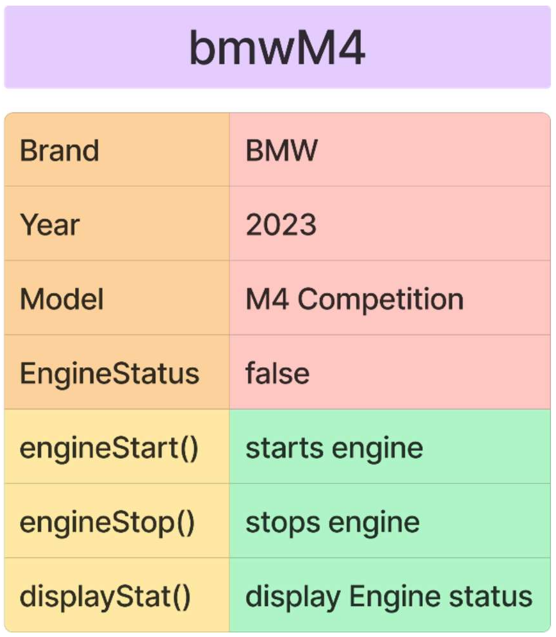
This creates a class and in the main method we create two objects:

```
public class CarDemo {
    public static void main(String[] args) {
        // Create an instance of the Car class
        Car porsche911 = new Car("Porsche", 2023, "911 GT3 RS");
        Car bmwM4 = new Car("BMW", 2023, "M4 Competition");
        porsche911.displayStats();
        bmwM4.displayStats();
    }
}
```



porsche911

Brand	Porsche
Year	2023
Model	911 GT3 RS
EngineStatus	false
engineStart()	starts engine
engineStop()	stops engine
displayStat()	display Engine status



bmwM4

Brand	BMW
Year	2023
Model	M4 Competition
EngineStatus	false
engineStart()	starts engine
engineStop()	stops engine
displayStat()	display Engine status

Constructors

A constructor helps to initialize an object (give values) immediately upon creation. In the previous example a constructor is used to initialize the data members but the other way is to access each of the data members separately using a period (.) operator and assigning them values.

```
porsche911.brand ="Porsche";
porsche911.model ="911 GT3 RS";
porsche911.year = 2023;
porsche911.engineStatus = false;
```

But the constructor is a method that returns nothing not even void because the implicit return type of the constructor is the class type itself, but is used to set initial conditions and values to the data members up on creation. Constructor is a special method inside the class. Constructor has the same name as the class in which it resides. Once defined, the constructor is automatically immediately called after the object is created, before the new operator completes. There are two types of constructors:

- Default Constructor – has no arguments
- Parameterized Constructor – has arguments

A default constructor is defined by

```
classname() {
      //body of constructor
}
```

These can be used to set some default values to the data members at the time of creation not letting them to be null thus causing troubles while executing some member methods. In the previous example let us consider that it has a default constructor

```
public Car() {
        this.brand = "Not branded";
        this.year = 0000;
        this.model = "Not registered";
        this.engineStatus = false;
}
```

This will initialise the given values if an object of Car class is made without passing any arguments. When we do not explicitly define a constructor for a class, then Java creates a default constructor for the class. For a new operator to create an object of a class it is necessary to have at least one constructor and thus Java creates one itself if there is no constructor.

A parameterized constructor can accept values using the parameter list that is passed on to it. Just as the one used in the previous example. The values are passed just like passing arguments to a method. Parameterized constructors are over loaded

this Keyword

In Java, the **this** keyword is a reference variable that refers to the current object. It is often used inside the methods or constructors of a class to refer to the current instance of the class. this() can be used to invoke current class constructor. this can be passed as an argument in the method call. this can be passed as argument in the constructor call. Static methods cannot refer to this as static members does not belong to objects but to the class.

We can have local variables, including formal parameters to methods, which has the same name of the class's instance variables(attributes). But when a local variable has the same name as an instance variable, the local variable hides the instance variable. This condition can be overcome using the this keyword as one of them refers to the formal parameters while the one with this keyword refers to the instance variable.

In the above example this approach was chosen to avoid instance variable hiding.

Method overloading

It is possible to define two or more methods with same name within the same class, but their parameter declarations should be different. Method overloading is a feature in Java that allows a class to have multiple methods with the same name but with different parameter lists within the same class. The methods must have either a different number of parameters or parameters of different types. Method overloading is a form of compile-time polymorphism, and it is also known as compile-time or static polymorphism.

Overloaded methods must differ in the type and/or number of their parameters. When an overloaded method is invoked, Java uses the type and/or number of arguments to determine which version of the overloaded method to actually call.

If in overloaded methods if there is a larger data type in the parameter list but the method invocation passes a smaller data type it is automatically converted to match the parameter list

Constructors are also overloaded so as to perform different initial logic while object creation.

Passing Parameters

There are two types of passing parameters:

- **Pass by value** – when primitive data types like int, char, double etc. are passed it is done by pass by value
- **Pass by reference** – when an object is passed to a function then it is done by pass by reference

Returning Objects

A method can return various types of data:

- Primitive data types like int, char, double etc.
- Class types like the object created by the user
- Void

An object can be returned by a method and this can be stored to another instance if needed.

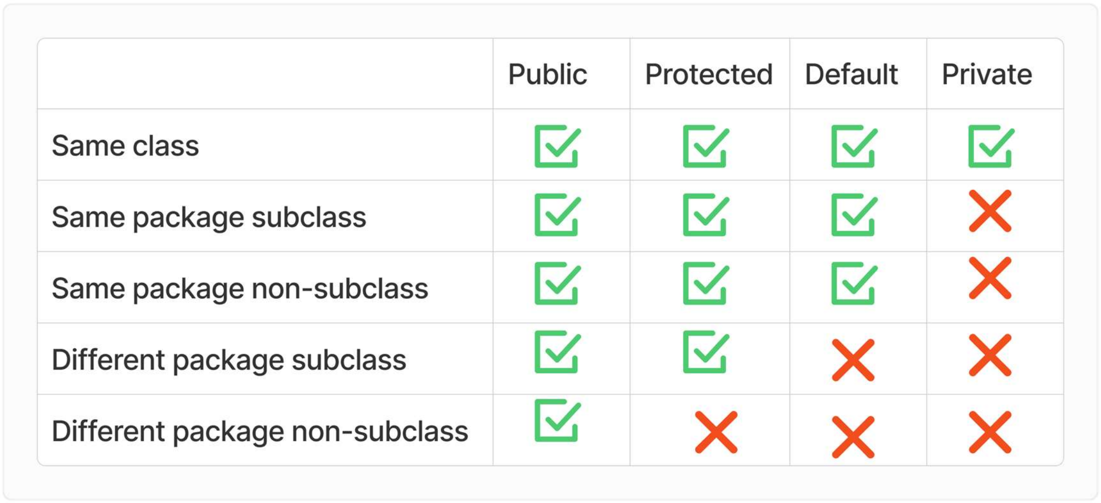
Recursion

Recursion is a programming concept in which a function calls itself in order to solve a problem. A recursive function consists of a base case and a recursive case. The base case is the condition under which the function stops calling itself, preventing an infinite loop. The recursive case represents the scenario in which the function calls itself to break down the problem into smaller subproblems.

Access Control

Access control refers to the mechanism that restricts the visibility and accessibility of classes, methods, and fields. It helps in enforcing encapsulation and protecting the integrity of a program. By controlling access, you can prevent misuse. There are four access specifiers in java

- **public** – accessible by any code and at any scope
- **private** – accessible only to the same class
- **protected** – accessible within the package and by any of its subclasses
- **default** – accessible only inside the package



	Public	Protected	Default	Private
Same class	✓	✓	✓	✓
Same package subclass	✓	✓	✓	✗
Same package non-subclass	✓	✓	✓	✗
Different package subclass	✓	✓	✗	✗
Different package non-subclass	✓	✗	✗	✗

Static Members

In Java, the static keyword is used to define members (fields and methods) that belong to the class itself rather than to instances of the class. These members are often referred to as "static members" or "class members." Unlike instance members, which are associated with objects created from the class, static members are associated with the class itself.

There are the two main types of static members in Java:

- static variable
- static methods

If we want to access a member of another class without using object, then we have to make it a make it a static member. When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object. It can be accessed using the

```
classname.member;
```

The most common example of a static member is main function. main( ) is declared as static because it must be called before any objects is created. Instance variables declared as static are global variables. When objects of its class are declared, separate copy of a static variable is NOT made and every one of them shares the same static member.

Static methods have some restrictions:

- static methods can only call other static methods.
- static methods must only access static data.
- static methods cannot refer to this or super.

If we need to do computation to initialize your static variables, we can declare a static block that gets executed exactly once, when the class is first loaded.

The static block is given as:

```
static {

	//body of block

}
```

Final Keyword

The **final** keyword serves as a powerful modifier, providing a means to denote constants, ensure method and class immutability, and prevent subclassing. When applied to variables, it signifies their unalterable values, often used for constants. For methods and classes, **final** ensures that their implementation or inheritance hierarchy remains unmodifiable, contributing to code security, robustness, and improved readability. It is a convention to choose uppercase identifiers for final variables. Variables declared as final do not occupy memory on a per instance basis.

Nested Classes

In Java, nested classes offer a powerful mechanism for organizing and encapsulating code within the context of another class. With four distinct types—**static nested classes**, **inner classes**, **local classes**, and **anonymous classes.** Java developers have flexible tools to structure their code based on specific design needs. Whether facilitating encapsulation, accessing outer class members, or enabling dynamic and localized implementations, the use of nested classes enhances the modularity and readability of Java programs. The scope of a nested class is bounded by the scope of its enclosing class(outer). A nested class has access to the members, including private members, of the enclosing(outer) class. The enclosing class do es not have access to the members of the nested class.

Static nested class

- A static nested class is associated with its outer class and doesn't have access to the instance-specific members of the outer class.
- It is declared using the static keyword.
- It must access the members of its enclosing class through an object.
- It cannot refer to members of its enclosing class directly.

Inner class

- An inner class is associated with an instance of its outer class and can access the instance-specific members of the outer class.
- It is a member of its enclosing class and is declared **without** the **static** keyword.
- An inner class has access and methods to all of the variables of its outer class.
- It may refer to members of its enclosing class directly in the same way that other non-static members of the outer class do.
- An instance(object) of Inner can be within the scope of class Outer.

We can create created only an instance of Inner class outside of Outer class by qualifying its name with Outer class name,

```
Outer.Inner ob=outerobject.new Inner();
```

Local class

- A local class is defined within a block of code, typically within a **method**.
- It has access to the local variables of the enclosing block and can also access instance variables of the outer class.

Anonymous class

An anonymous class is a local class without a name, often used for one-time use, such as event handling. It is declared and instantiated at the same time.

```
OuterClass.StaticNestedClass staticNestedObj = new
OuterClass.StaticNestedClass();
```

Is used to instantiate a static nested class

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

Is used to instantiate an inner class

Command Line Arguments

If we want to pass information into a program when you run it, then you can do this by passing command-line arguments to main( ). A command-line argument is the information that follows program's name on the command line when it is executed. The arguments passed on are mapped to the String[] args variable in the main function parameter list.

The first command-line argument is stored at args[0]. The general form is

```
java fileName argument1 argument2 … ;
```

```
or
```

```
java fileName argument1,argument2, … ;
```

Variable length arguments

Variable-length argument lists, also known as varargs, provide a flexible feature in Java that allows methods to accept a variable number of arguments of the same type. Introduced with Java 5, varargs simplify method calls by enabling developers to pass an arbitrary number of arguments without explicitly defining each parameter. Declared using an ellipsis (**...**) in the method signature, varargs enhance the versatility and adaptability of methods, making them particularly useful in scenarios where the number of arguments may vary.

- If the **maximum number** of arguments is **small** and **known**, then we can create overloaded versions of the method, one for each way the method could be called.
- If the **maximum number** of potential arguments is **larger**, or unknowable, then the arguments can be put into an **array**, and then the array can be passed to the method.

```
returnType methodName(type... parameterName) {

    // Method implementation

}
```

When using variable-length argument lists (varargs) in Java, there are several rules and considerations to keep in mind:

- The varargs parameter (the one with the ellipsis **...**) must be the last parameter in the method's parameter list.
- A method can have at most one varargs parameter.
- The varargs parameter must be of an array type.

- Varargs can be used with overloaded methods, but care must be taken to avoid ambiguity.

Varargs and Ambiguity

In Java, ambiguity related to varargs occurs when there is potential confusion or uncertainty in the method invocation due to overloading and the presence of varargs. This ambiguity arises when the compiler has difficulty determining which overloaded method to invoke based on the arguments provided.

Consider two overloaded method declaration:

```
static void test(int ... v) { // ... }

static void test(int n, int ... v) { // ... }
```

calling test() with just one variable will create an ambiguity

Inheritance

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class (the subclass or derived class) to inherit attributes and behaviours from another class (the superclass or base class). This relationship enables code reuse, promotes modularity, and facilitates the creation of a hierarchy of classes. Inheritance establishes an "is-a" relationship between classes, where a subclass is a specialized version of its superclass, inheriting its properties while potentially adding or modifying them.

In Java, inheritance is implemented using the **extends** keyword, allowing a subclass to extend a superclass. The inherited members include fields, methods, and nested classes, providing a mechanism for building upon existing code and promoting a more organized and efficient software design.

Subclass cannot access the private members in superclass. A major advantage of inheritance is that once you have created a superclass that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses. Each sub class can have its own special features also.

A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.

Super Keyword

In Java, the **super** keyword plays a crucial role in the realm of inheritance, allowing subclasses to access and leverage the features of their superclass. By using **super**, developers can navigate potential naming conflicts, access overridden members of the superclass, and invoke the superclass's constructor to ensure proper initialization. This keyword contributes to the flexibility and extensibility of Java code, facilitating the creation of well-organized and reusable class hierarchies. The keyword is used for two purposes:

- Call the superclass constructor by super( )
- Access members of super class that has been hidden by any of the member of subclass

The static methods cannot refer to super. super( ) must always be the first statement executed inside a subclass’s constructor.

Protected members

In Java, the protected access modifier is used to declare members (fields, methods, and nested classes) that are accessible within the same package, as well as by subclasses, regardless of whether they are in the same package or a different one.

Calling order of Constructors

Constructors are called in the order of derivation, from superclass to **subclass.** When subclass object is created, it first calls superclass constructor then only it calls subclass constructor. If super( ) is not used to call superclass constructor, then the default constructor of each superclass will be executed before executing subclass constructors. Superclass has no knowledge of any subclass, any initialization it needs to perform is separate and it should be done as a prerequisite to initialize the subclass object.

Method Overriding

Method overriding occurs when a subclass provides its own implementation for a method that is already present in its superclass. The signature (name, return type, and parameter types) of the overridden method must be the same as that of the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the method defined by the subclass. The version of the method defined by the superclass will be hidden.

To access the superclass version of an overridden method, we can use the super keyword.

Object Class

The **Object** class stands as the cornerstone of Java's class hierarchy, serving as the root from which all other classes are derived. Implicitly inherited by every Java class, it equips them with fundamental methods such as **toString()**, **equals()**, **hashCode()**, **getClass()**, and **finalize()**. While these methods provide default behaviors, they are often overridden in subclasses to tailor functionality to specific needs, making the **Object** class a crucial component for understanding and customizing the behavior of Java objects. It plays a foundational role in object-oriented programming, encapsulating essential functionalities and fostering consistent interactions across diverse classes in Java applications.

All other classes are subclasses of **Object**. Reference variable of type Object can refer to an object of any other class.

The methods **getClass( ), notify( ), notifyAll( ),** and **wait( )** are declared as **final**. The **equals( )** method compares the contents of two objects. The **toString( )** method returns a string that contains a description of the object on which it is called.

Abstract classes and Methods

Sometimes we may want to create a superclass that only defines a generalized form which will be shared by all of its subclasses and leaves the implementation to be filled by each subclass.

To ensure that a subclass should override all necessary methods(implementations), we have to make them abstract methods in superclass. For making a method an abstract method we have use abstract type modifier.

Abstract methods have no implementation (function body) in the superclass so they are also called as subclasser responsibility. The implementation should be there in subclasses by overriding those methods.

```
abstract type name(parameter-list);
```

here instead of curly braces a semi colon is used to define it's a method with no implementation.

Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, use the abstract keyword in front of the class keyword at the beginning of the class declaration.

```
abstract class className {

      //members, abstract or concrete methods

}
```

Abstract classes cannot be instantiated using new operator that is, we can't create instances of an abstract class. Such objects would be useless, because an abstract class is not fully defined.

Any subclass of an abstract class must either of implement all the abstract methods in the superclass, or it should be **declared abstract class.**

Although abstract classes cannot be used to instantiate objects, abstract classes can be used to create object references. Java's run-time polymorphism(dynamic binding) is implemented through the use of superclass references.

Final Keyword in Inheritance

The final keyword helps to prevent inheritance and over riding. If we don't want to allow subclass to override a method of superclass's method, we can use final as a modifier at the start of its declaration in superclass. Methods declared as final cannot be overridden by subclass. Methods declared as final can sometimes provide a performance enhancement The compiler is free call them inline because it "knows" they will not be overridden by a subclass.