Contents

[OOPS concepts 4](#_Toc188643631)

[Encapsulation: 4](#_Toc188643632)

[Abstraction: 4](#_Toc188643633)

[Key Differences: 4](#_Toc188643634)

[Memory Allocation in Java 5](#_Toc188643635)

[Static Imports 6](#_Toc188643636)

[Constructors 7](#_Toc188643637)

[Default constructors 7](#_Toc188643638)

[This keyword 7](#_Toc188643639)

[The static Keyword in Java 9](#_Toc188643640)

[Static Initialization Blocks 11](#_Toc188643641)

[Order of Initialization 13](#_Toc188643642)

[Abstraction in java 14](#_Toc188643643)

[Abstract Class: 14](#_Toc188643644)

[Interface: 15](#_Toc188643645)

[Key Differences Between Abstract Class and Interface: 16](#_Toc188643646)

[Arrays methods 17](#_Toc188643647)

[Arrays.sort(); 17](#_Toc188643648)

[Arrays.equals() 17](#_Toc188643649)

[Arrays.fill() 17](#_Toc188643650)

[Arrays.asList() 18](#_Toc188643651)

[Arrays.toString() 18](#_Toc188643652)

[ArrayList 19](#_Toc188643653)

[Arrays and ArrayList conversion 19](#_Toc188643654)

[Raw Types vs. Generics in ArrayList 20](#_Toc188643655)

[Example Comparison 20](#_Toc188643656)

[Summary 21](#_Toc188643657)

[concurrent modification error 21](#_Toc188643658)

[Iterating Over ArrayList 21](#_Toc188643659)

[Using left shift operator to find the power of 2; 23](#_Toc188643660)

[Scanner 24](#_Toc188643661)

[scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?"); 24](#_Toc188643662)

[Scanner.hasNextLine(); 24](#_Toc188643663)

[BufferReader 25](#_Toc188643664)

[Range of primitive data types 26](#_Toc188643665)

[Assigning literals to variables 27](#_Toc188643666)

[Primitive char and Unicode characters 29](#_Toc188643667)

[Default Values Assigned to Primitive Data Types 30](#_Toc188643668)

[Wrapper Classes 31](#_Toc188643669)

[Boxing and Autoboxing 31](#_Toc188643670)

[Assignment operator 32](#_Toc188643671)

[Short circuiting 33](#_Toc188643672)

[== and equals method 34](#_Toc188643673)

[The switch case conflict 35](#_Toc188643674)

[Static initialization blocks 36](#_Toc188643675)

[Strings 38](#_Toc188643676)

[String to int and int to String 40](#_Toc188643677)

[Integer.parseInt() 40](#_Toc188643678)

[String pool and string Equality 41](#_Toc188643679)

[Constant folding at compile time 42](#_Toc188643680)

[Runtime Concatenation 42](#_Toc188643681)

[StringBuilder 43](#_Toc188643682)

[Ways to Specify Capacity: 44](#_Toc188643683)

[What Happens Internally: 45](#_Toc188643684)

[Why Strings are immutable in java? 46](#_Toc188643685)

[1. Security: 46](#_Toc188643686)

[2. Caching and Performance (String Pool): 46](#_Toc188643687)

[3. Hashcode Consistency: 46](#_Toc188643688)

[4. Design Simplicity: 46](#_Toc188643689)

[5. Security in Class Loading and Reflection: 46](#_Toc188643690)

[Date and time 47](#_Toc188643691)

[Methods 47](#_Toc188643692)

[BigDecimal 48](#_Toc188643693)

[BigInteger 49](#_Toc188643694)

[To check if all digits of a number are pairwise different 51](#_Toc188643695)

[Why we use srand in c++; 51](#_Toc188643696)

[Abstract method 51](#_Toc188643697)

[Lambda Expression 53](#_Toc188643698)

[Interface 54](#_Toc188643699)

[File Handling 55](#_Toc188643700)

[Collection Framework 57](#_Toc188643701)

[1. Iterable Interface 57](#_Toc188643702)

[2. Collection Interface 57](#_Toc188643703)

[Difference between arraylist and vectors 58](#_Toc188643704)

[Set in Java 60](#_Toc188643705)

[Map 61](#_Toc188643706)

[Streams 62](#_Toc188643707)

[Exceptions in Java 64](#_Toc188643708)

[Executors Framework in java 66](#_Toc188643709)

[CompletableFuture 68](#_Toc188643710)

[Java 8 69](#_Toc188643711)

[Lambda Expression 69](#_Toc188643712)

[Functional Interface 71](#_Toc188643713)

[Default methods inside interface 71](#_Toc188643714)

[Static methods inside interface 71](#_Toc188643715)

[Date-Time API 73](#_Toc188643716)

[Resources 74](#_Toc188643717)

# OOPS concepts

The four main concepts of Java Object-Oriented Programming (OOP) are:

1. **Encapsulation**: This is the concept of wrapping data (variables) and methods (functions) that operate on the data into a single unit, or class. It restricts direct access to some of an object's components, which is why we use getters and setters. This protects the internal state of the object and helps in achieving data hiding.
2. **Inheritance**: Inheritance allows a new class (subclass) to inherit properties and behaviors (methods) from an existing class (superclass). This promotes code reusability and helps in establishing a relationship between different classes (parent-child relationship). For example, a Dog class can inherit from an Animal class.
3. **Polymorphism**: Polymorphism allows methods or functions to take on multiple forms. It comes in two types:
   * **Compile-time polymorphism** (method overloading): Multiple methods in the same class with the same name but different parameters.
   * **Runtime polymorphism** (method overriding): A subclass provides a specific implementation of a method that is already defined in its superclass.
4. **Abstraction**: Abstraction hides the complex implementation details and shows only the necessary features of an object. In Java, abstraction is achieved using abstract classes and interfaces. It allows focusing on what an object does rather than how it does it. For example, a Vehicle interface might have an abstract method start(), which is implemented differently by Car and Bike classes.

**Difference between encapsulation and abstraction**

### **Encapsulation**:

* **Definition**: Encapsulation is the practice of bundling the data (attributes) and methods (functions) that operate on that data into a single unit or class. It restricts direct access to certain components of an object and protects its internal state from unintended modifications.
* **Focus**: Encapsulation is concerned with controlling access to the data within an object.
* **How it works**: It achieves data hiding by making variables private or protected and providing public getter and setter methods to manipulate or access the data safely.

### **Abstraction**:

* **Definition**: Abstraction is the concept of hiding complex implementation details and showing only the essential features of an object. It focuses on "what" an object does rather than "how" it does it.
* **Focus**: Abstraction is concerned with hiding unnecessary details and providing a simple interface for interaction.
* **How it works**: It is achieved through abstract classes and interfaces that define methods without implementation. The concrete classes provide the specific implementation.
* **Example**: A Vehicle interface has an abstract method start(). The concrete classes Car and Bike implement this method differently.

### Key Differences:

* **Encapsulation**: Focuses on **how** data is accessed and protected, using access modifiers to control visibility (e.g., private, public).
* **Abstraction**: Focuses on **what** functionality an object provides, hiding complex logic through abstract methods or interfaces.

# Memory Allocation in Java

1. **Heap Memory:**
   * **Arrays:** Arrays, whether they are arrays of primitive types or arrays of objects, are allocated on the heap. The heap is a region of memory used for dynamic memory allocation, and it is managed by the Java Virtual Machine (JVM). The heap memory is shared among all threads.
   * **Objects:** All objects are also allocated on the heap. This includes instances of user-defined classes.
2. **Stack Memory:**
   * **Local Variables:** Local variables (including primitives and references to objects) declared within methods are stored on the stack. Each thread has its own stack, which is used to store frames. Each frame contains local variables, the operand stack, and the frame data.
   * **Method Call Frames:** When a method is called, a new frame is created on the stack for that method call. This frame contains all the local variables and the call's execution state.

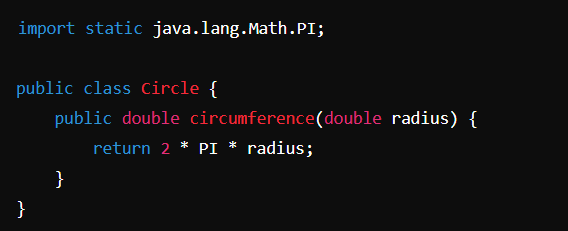
**Detailed Explanation**

* **Heap Memory:**
  + Heap memory is used for dynamic memory allocation where objects and arrays reside. This memory is managed by the garbage collector, which automatically reclaims memory that is no longer in use.
  + Arrays in Java are considered objects. When you create an array, the JVM allocates memory for the array on the heap and initializes it to the default values for the array's element type.
* **Stack Memory:**
  + Stack memory is used for method execution and contains method-specific data such as local variables and partial results. Each time a method is invoked, a new frame is pushed onto the stack.
  + Local variables within a method are stored in the stack frame of that method. These local variables can be primitive types or references to objects (the actual objects reside on the heap).

In Java, class variables (also known as **static variables**) are not stored in the heap. They are stored in a special area of memory called the **method area**, which is part of the **Java Virtual Machine (JVM)**. The method area holds class-level data, including static variables, method bytecodes, and other metadata related to classes.

# Static Imports

In Java, static import is a feature that allows members (fields and methods) defined in a class to be used in another class without specifying the class they belong to.



Only static members of a class can be imported using static imports . we cannot import non static members with this.  
  
  
  
Autoboxing and Unboxing in Java

In Java, primitive data types are treated differently so do there comes the introduction of wrapper classes where two components play a role namely Autoboxing and Unboxing. Autoboxing refers to the conversion of a primitive value into an object of the corresponding wrapper class is called autoboxing. For example, converting int to Integer class. The Java compiler applies autoboxing when a primitive value is:

Passed as a parameter to a method that expects an object of the corresponding wrapper class.

Assigned to a variable of the corresponding wrapper class.

Unboxing on the other hand refers to converting an object of a wrapper type to its corresponding primitive value. For example conversion of Integer to int. The Java compiler applies to unbox when an object of a wrapper class is:

Passed as a parameter to a method that expects a value of the corresponding primitive type.

Assigned to a variable of the corresponding primitive type.

# Constructors

## Default constructors

If we do not create a constructor then java compiler creates a no argument constructor for us by default.

We can create multiple constructors  
if make only one default constructor and make it private then that class cannot be instantiated ….this is epful in for classes whose all methos are abstract.

## This keyword

The this keyword is a reference variable in Java that points to the current object. It is primarily used within a class to differentiate between instance variables and parameters, invoke constructors, and pass the current object as an argument. Here’s a breakdown of its usage:

**1. Referencing Instance Variables:**

The this keyword is used to resolve naming conflicts between instance variables and parameters. When an instance variable has the same name as a parameter, this helps clarify that the instance variable is being referenced.

public class Person {  
 String firstName;  
 String lastName;  
  
 public Person(String firstName, String lastName) {  
 // Using `this` to refer to instance variables  
 this.firstName = firstName;  
 this.lastName = lastName;  
 }  
}

**2. Calling Another Constructor:**

The this() keyword is used to call another constructor within the same class. This is useful for constructor chaining, where one constructor calls another constructor to reuse code.

public class Person {  
 String firstName;  
 String lastName;  
  
 // Default constructor calls parameterized constructor  
 public Person() {  
 this("Anannya", "Hiteshi");  
 }  
  
 // Parameterized constructor  
 public Person(String firstName, String lastName) {  
 this.firstName = firstName;  
 this.lastName = lastName;  
 }  
}

Key Points:

The this() call must be the first statement in the constructor.

It allows constructors to reuse code and reduce redundancy.

**3. Passing the Current Object:**

The this keyword can be used to pass the current object as an argument to methods or constructors.

public class Person {  
 String firstName;  
 String lastName;  
  
 public Person(String firstName, String lastName) {  
 this.firstName = firstName;  
 this.lastName = lastName;  
 }  
  
 // Method that takes Person object as parameter  
 public void printPerson(Person p) {  
 System.*out*.println("First Name: " + p.firstName);  
 System.*out*.println("Last Name: " + p.lastName);  
 }  
  
 // Passing current object using `this`  
 public void display() {  
 printPerson(this);  
 }  
}

**4. Returning the Current Object:**

The this keyword can be used to return the current instance from a method, allowing method chaining.

public class Person {  
 String firstName;  
 String lastName;  
  
 public Person setFirstName(String firstName) {  
 this.firstName = firstName;  
 return this; // returning current object  
 }  
  
 public Person setLastName(String lastName) {  
 this.lastName = lastName;  
 return this;  
 }  
}

# The static Keyword in Java

The static keyword in Java is used to indicate that a particular member (variable, method, or nested class) belongs to the class itself rather than to instances of the class. This means that:

1. **Static Variables:**
   * A static variable is shared among all instances of the class. There is only one copy of the static variable in memory, regardless of how many instances of the class are created.
   * Static variables can be accessed directly using the class name (e.g., ClassName.variableName), without needing an instance of the class.
2. class Example {  
    static int *counter* = 0;  
     
    public Example() {  
    *counter*++; // Incrementing the static variable  
    }  
   }  
     
   class Test {  
    public static void main(String[] args) {  
    Example obj1 = new Example();  
    Example obj2 = new Example();  
    System.*out*.println(Example.*counter*); // Outputs: 2  
    }  
   }
3. **Static Methods:**
   * Static methods can be called without creating an instance of the class. They can only directly access other static members (variables and methods) of the class.
4. class MathUtils {  
    static int square(int number) {  
    return number \* number;  
    }  
   }  
     
   class Test {  
    public static void main(String[] args) {  
    int result = MathUtils.*square*(5); // Calling static method  
    System.*out*.println(result); // Outputs: 25  
    }  
   }
5. **Static Blocks:**
   * Static blocks are used to initialize static variables. They are executed when the class is loaded into memory, before any objects of the class are created and before any static methods are called.

# The static keyword can also be applied to a class but only when it is nested inside another class. If we use static keyword for a top level class then an error will be thrown

Example:-

public class ThreadExample2 {  
 public static class MyRunnable implements Runnable {  
 public void run() {  
 System.*out*.println("MyRunnable");  
 }  
 }  
   
 public static void main(String[] args) {  
 Thread thread = new Thread(new MyRunnable());  
 thread.start();  
 }  
}

If you would not make the nested class static then in that case you will have to create an object of the outer class within the main method to create the MyRunnable obj

public class ThreadExample2 {  
 public class MyRunnable implements Runnable{  
 public void run() {  
 System.*out*.println("MyRunnable");  
 }  
 }  
 public static void main(String[] args) {  
 ThreadExample2 outer = new ThreadExample2();  
  
 Thread thread = new Thread(outer.new MyRunnable());  
 thread.start();  
 }  
}

# Static Initialization Blocks

Static initialization blocks, also known as static blocks, are used for initializing static variables or performing other setup tasks at the time of class loading. These blocks run only once, when the class is first loaded into memory.

**Characteristics of Static Blocks:**

* Multiple static blocks can be used in a class, and they are executed in the order they appear in the code.
* Static blocks are executed before the main method or any constructors of the class.

**Example of Static Initialization Blocks:**

public class Example {  
 static int *value*;  
  
 // First static block  
 static {  
 *value* = 5;  
 System.*out*.println("First static block: value = " + *value*);  
 }  
  
 // Second static block  
 static {  
 *value* = 10;  
 System.*out*.println("Second static block: value = " + *value*);  
 }  
  
 public static void main(String[] args) {  
 System.*out*.println("Main method: value = " + *value*);  
 }  
}

**Explaination:**

1. **Static Blocks Execution:**
   * When the Example class is loaded, the static blocks are executed in the order they appear.
   * The first block sets value to 5 and prints the message.
   * The second block then sets value to 10 and prints a new message.
2. **Output:**
   * The static blocks run before the main method, so the value of value when the main method is executed is 10, the last value assigned by the static blocks.

**Output:**

sql  
Copy code  
First static block: value = 5  
Second static block: value = 10  
Main method: value = 10

**Usage:**

* Static blocks are useful for initializing static variables with complex logic or for performing setup tasks that need to happen when the class is first loaded.
* However, static blocks should be used carefully as they run before the class is fully initialized and can make the code harder to read if overused.

# Order of Initialization

1. If there is superclass initialize it first.
2. Static variable declarations and static initializers in order they appear in the file
3. Instance variable declarations and instance initializers in order they appear in the file
4. The constructor.

# Abstraction in java

### **Abstract Class**:

An **abstract class** in Java is a class that cannot be instantiated on its own. It is used as a blueprint for other classes and may contain abstract methods (methods without a body) as well as concrete methods (with a body). Abstract classes provide partial abstraction and are typically used when classes share some common behavior but also need some specific implementation.

#### Key Features of Abstract Class:

1. **Partial abstraction**: It can have both abstract (without implementation) and non-abstract (with implementation) methods.
2. **Constructor and instance variables**: Abstract classes can have constructors, member variables, and methods like regular classes.
3. **Inheritance**: A class extends an abstract class using the extends keyword. The subclass must implement all abstract methods or be declared abstract itself.
4. **Single inheritance**: A class can inherit only one abstract class (Java doesn’t support multiple inheritance with classes).

#### Example:

java

Copy code

abstract class Animal {

// Abstract method (no body)

public abstract void sound();

// Concrete method

public void sleep() {

System.out.println("Animal is sleeping");

}

}

class Dog extends Animal {

public void sound() {

System.out.println("Dog barks");

}

}

class Cat extends Animal {

public void sound() {

System.out.println("Cat meows");

}

}

public class Main {

public static void main(String[] args) {

Animal dog = new Dog();

dog.sound(); // Outputs: Dog barks

dog.sleep(); // Outputs: Animal is sleeping

}

}

In this example, Animal is an abstract class, and the concrete classes Dog and Cat implement the abstract method sound().

### **Interface**:

An **interface** in Java is a completely abstract type that is used to specify a set of methods that a class must implement. It contains only abstract methods (before Java 8) and constants. Interfaces are used to achieve full abstraction and to define the behavior that a class must implement.

#### Key Features of Interface:

1. **Full abstraction**: Before Java 8, all methods in an interface are abstract (without implementation). Since Java 8, it can also have default methods (with a body) and static methods.
2. **No constructors or instance variables**: Interfaces cannot have constructors or instance variables; only constant variables (final and static).
3. **Multiple inheritance**: A class can implement multiple interfaces, thus achieving multiple inheritance in Java.
4. **Implementation**: A class implements an interface using the implements keyword and must provide implementations for all abstract methods of the interface.

#### Example:

java

Copy code

interface Vehicle {

void start(); // abstract method

}

interface FuelVehicle {

void refuel();

}

class Car implements Vehicle, FuelVehicle {

public void start() {

System.out.println("Car starts");

}

public void refuel() {

System.out.println("Car is refueled");

}

}

public class Main {

public static void main(String[] args) {

Car myCar = new Car();

myCar.start(); // Outputs: Car starts

myCar.refuel(); // Outputs: Car is refueled

}

}

In this example, Vehicle and FuelVehicle are interfaces, and Car implements both of them. This demonstrates multiple inheritance, where a class can implement more than one interface.

### Key Differences Between Abstract Class and Interface:

| **Feature** | **Abstract Class** | **Interface** |
| --- | --- | --- |
| **Methods** | Can have both abstract and concrete methods | Only abstract methods (Java 8 introduced default/static) |
| **Constructors** | Can have constructors | Cannot have constructors |
| **Variables** | Can have instance variables and constants | Can only have static and final variables |
| **Multiple inheritance** | A class can extend only one abstract class | A class can implement multiple interfaces |
| **Use case** | When classes share common functionality but need some specific implementations | When unrelated classes need to share a common behavior |

In short, use **abstract classes** when you need to share behavior between closely related classes, and use **interfaces** when you want to specify a contract that can be applied to unrelated classes.

# Arrays methods

## Arrays.sort();

* The complexity of the **Arrays.sort(a)** method in Java depends on the sorting algorithm used by the implementation. The **Arrays.sort(a)** method typically uses a modified version of the Quicksort algorithm for primitive types and a modified version of the TimSort algorithm for objects.
* For primitive types (**int**, **char**, etc.), the **Arrays.sort(a)** method uses a dual-pivot Quicksort algorithm, which has an average-case time complexity of O(n log n), where n is the number of elements in the array. In the worst case, the dual-pivot Quicksort algorithm has a time complexity of O(n^2), but this is very rare and occurs only for specific input patterns.
* For objects, the **Arrays.sort(a)** method uses a TimSort algorithm, which is a hybrid sorting algorithm combining merge sort and insertion sort. The TimSort algorithm has an average-case time complexity of O(n log n) and guarantees stable sorting (i.e., the relative order of equal elements is preserved).
* In both cases, the space complexity of the **Arrays.sort(a)** method is O(log n), indicating that it requires additional memory for recursion or temporary storage during the sorting process.
* It's worth noting that the actual performance of **Arrays.sort(a)** can vary depending on factors such as the size of the array, the data distribution, and the specific implementation used by the Java runtime environment. However, in general, it provides an efficient and reliable way to sort arrays in Java.

## Arrays.equals()

This method is used to compare the elements of two arrays to check if they contain the same elements  
  
Example : Arrays.equals(arr1, arr2)

## Arrays.fill()

This method is used to fill an array with a given value  
  
Example : int arr[] = new int[4];

Arrays.fill(arr, 1)

The new array will be : [1,1,1,1]

## Arrays.asList()

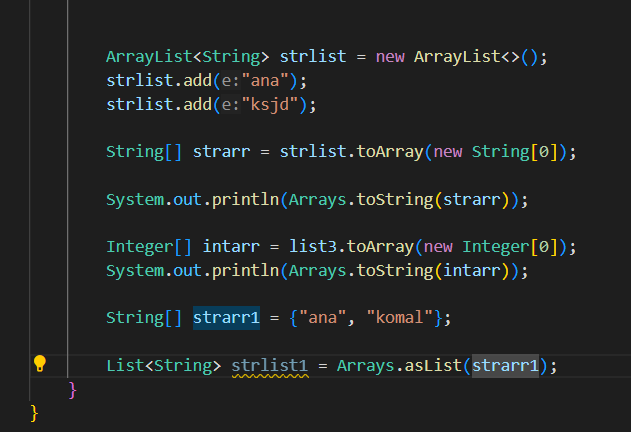
This method is used to return an array as a list …basically convert an array to a list  
  
List<Integer> list = Arrays.asList(arr)

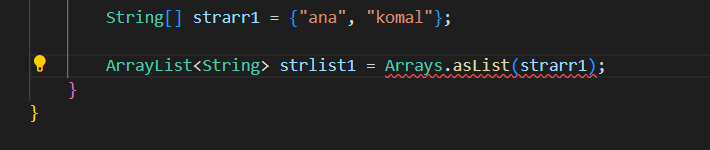
# Arrays.toString()

System.out.println(Arrays.toString(arrName)); //prints the whole array as => [element1, element2, …]

# ArrayList

## Arrays and ArrayList conversion





It looks like there's a type mismatch in your code. The Arrays.asList method returns a List, not an ArrayList, so you cannot directly assign it to an ArrayList variable.

## Raw Types vs. Generics in ArrayList

#### Raw Types

When you use raw types, you don't specify the type of objects the collection should contain. For example:

java

Copy code

ArrayList list1 = new ArrayList();

list1.add("Hello");

list1.add(1); // Adding an integer

With raw types, the compiler cannot enforce type constraints. As a result, you can mix different types within the same collection, which can lead to issues when you later try to retrieve and cast these elements.

Example of potential runtime error:

java

Copy code

ArrayList list1 = new ArrayList();

list1.add("Hello");

list1.add(1);

String s = (String) list1.get(1); // This will throw ClassCastException at runtime

In the above example, retrieving an Integer from the list and casting it to a String will result in a ClassCastException at runtime.

#### Generics

Generics provide a way to specify the type of objects that a collection can contain, thereby enforcing type safety at compile time. For example:

java

Copy code

ArrayList<String> list1 = new ArrayList<>();

list1.add("Hello");

// list1.add(1); // Compile-time error: incompatible types

Here, ArrayList<String> specifies that the list can only contain String objects. The compiler enforces this constraint, so attempting to add an Integer will result in a compile-time error rather than a runtime error.

### Example Comparison

**Using Raw Types:**

java

Copy code

ArrayList list1 = new ArrayList();

list1.add("Hello");

list1.add(1);

String s = (String) list1.get(1); // Runtime error: ClassCastException

**Using Generics:**

java

Copy code

ArrayList<String> list1 = new ArrayList<>();

list1.add("Hello");

// list1.add(1); // Compile-time error: incompatible types

### Summary

* **Raw Types**: Allow any type of object, which can lead to runtime errors if objects are incorrectly cast or used.
* **Generics**: Enforce type safety at compile time, reducing the risk of runtime errors related to type mismatches.

By using generics, you ensure that only objects of the specified type can be added to the collection, catching type errors early during compilation and preventing potential runtime issues.

# 

## concurrent modification error

We can not iterate and modif a list at same time foreac loop gives concurrent modification error   
  
  
Iterating ArrayList

### Iterating Over ArrayList

**1. For Loop**

for (int i = 0; i < list.size(); i++) {

        Integer number = list.get(i);

        System.out.println(number);

    }

**2. Enhanced For Loop**

  for (Integer number : list) {

        System.out.println(number);

    }

**3. Iterator**

 for(Iterator<Integer> iterator = list3.iterator(); iterator.hasNext();){

            Integer number = iterator.next();

            System.out.println();

            iterator.remove();

        }

**How Iterator Works:**

* **Initialization**: Iterator is created with list.iterator(), initially pointing before the first element.
* **Traversal**: iterator.next() moves the cursor to the next element and returns it. Calling next() advances the cursor.
* **Removal**: iterator.remove() safely removes the last element returned by next(). This avoids ConcurrentModificationException, which can occur in normal loops if the list is modified directly during iteration.

**Notes:**

* **Iterator**: Safely handles concurrent modification when removing elements during iteration, unlike traditional loops.

When it is said that the Iterator "points between elements," it means:

* **Initial State**: When an Iterator is first created, it is positioned before the first element of the collection. It does not point to any specific element yet.
* **During Iteration**: As you call iterator.next(), the iterator moves to the next element. After calling next(), the iterator points to the element just returned and is positioned between this element and the next one.

# Using left shift operator to find the power of 2;

The shift operators in Java (**<<**, **>>**, and **>>>**) are used for shifting the bits of a number to the left or right. In the context of calculating powers of 2, the left shift operator (**<<**) is commonly used.

The left shift operator (**<<**) shifts the bits of a number to the left by a specified number of positions. Each shift to the left effectively multiplies the number by 2. Here's how it works:

1. Start with the binary representation of a number that is a power of 2. For example, **1** is **0001**, **2** is **0010**, **4** is **0100**, and so on.
2. Applying the left shift operator (**<<**) to a power of 2 shifts all the bits to the left by the specified number of positions. For example, **1 << 1** shifts **0001** one position to the left, resulting in **0010**, which is the binary representation of **2**. Similarly, **1 << 2** shifts **0001** two positions to the left, resulting in **0100**, which is the binary representation of **4**.

By using the left shift operator repeatedly, you can calculate higher powers of 2. For example, **1 << 3** would give **1000**, which is the binary representation of **8**.

In the updated code snippet provided earlier, **(1 << temp)** is used to calculate the value of **2** raised to the power of **temp**. It effectively performs a left shift operation on the binary representation of **1** by **temp** positions, which results in the binary representation of the corresponding power of 2.

By using the left shift operator (**<<**), the code can efficiently check if a number is divisible by a power of 2 and calculate the largest power of 2 that divides the given number **x**.

# Scanner

## scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");

The code snippet you provided seems to be written in Java.

The line **scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");** is using the **skip** method of a **Scanner** object. The regular expression **(\r\n|[\n\r\u2028\u2029\u0085])?** is passed as the argument to the **skip** method.

This regular expression is used to skip any line terminators or newlines present in the input stream being scanned. The line terminators it matches are:

* **\r\n**: Carriage return followed by a newline (Windows style)
* **[\n\r\u2028\u2029\u0085]**: Any of the characters newline, carriage return, line separator, paragraph separator

The **?** at the end of the regular expression makes the entire expression optional, meaning it will skip zero or one occurrence of the line terminators or newlines.

In summary, this line of code is used to skip over any line terminators or newlines present in the input stream being scanned by the **Scanner** object

## Scanner.hasNextLine();

To read an unknown number of lines of input from **stdin** (System.in) until reaching the end of file (EOF), you can use a **Scanner** object and a while loop. The loop will continue reading input as long as there is more input available.

Here's an example code snippet that demonstrates reading input lines until reaching EOF:

import java.util.Scanner;  
  
public class Solution {  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.*in*);  
  
 while (scanner.hasNextLine()) {  
 String line = scanner.nextLine();  
 // Process each line of input here  
 System.*out*.println("Input line: " + line);  
 }  
  
 scanner.close();  
 }  
}

In this code, the **hasNextLine()** method of the **Scanner** object is used to check if there is another line of input available. If there is, the **nextLine()** method is called to read the entire line as a **String**. You can process each line of input within the loop according to your requirements.

The loop will continue until there are no more lines of input available, indicating that the end of file (EOF) has been reached.

Note that after finishing reading input, it's good practice to close the **Scanner** object using the **close()** method to release any associated resources.

Feel free to modify the code inside the loop to perform the desired operations on each line of input.

# BufferReader

BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(System.in));

        int N = Integer.parseInt(bufferedReader.readLine().trim());

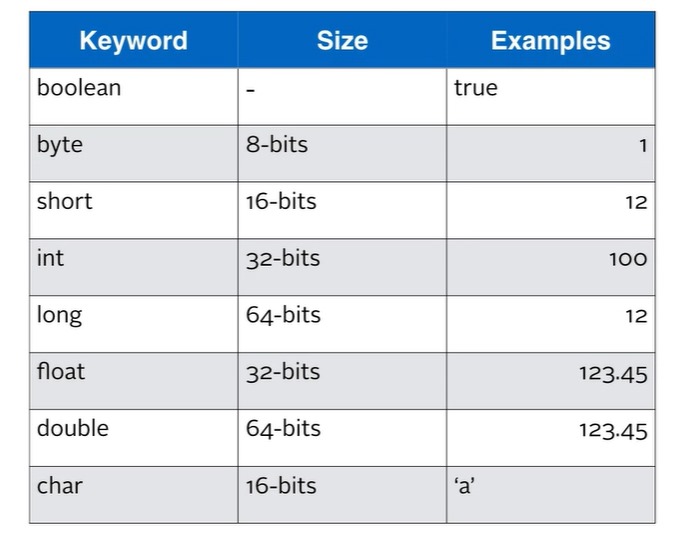
        bufferedReader.close();

# Range of primitive data types

Java has 8 primitive data types; char, boolean, byte, short, int, long, float, and double. For this exercise, we'll work with the primitives used to hold integer values (byte, short, int, and long):

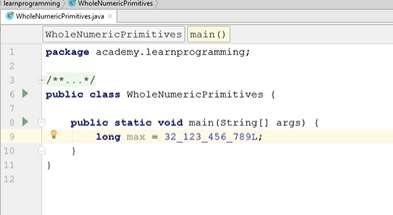
* A byte is an 8-bit signed integer. -128 to 127
* A short is a 16-bit signed integer. -32,768 to 32,767
* An int is a 32-bit signed integer. -2,147,483,648 to 2,147,483,647
* A long is a 64-bit signed integer. -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

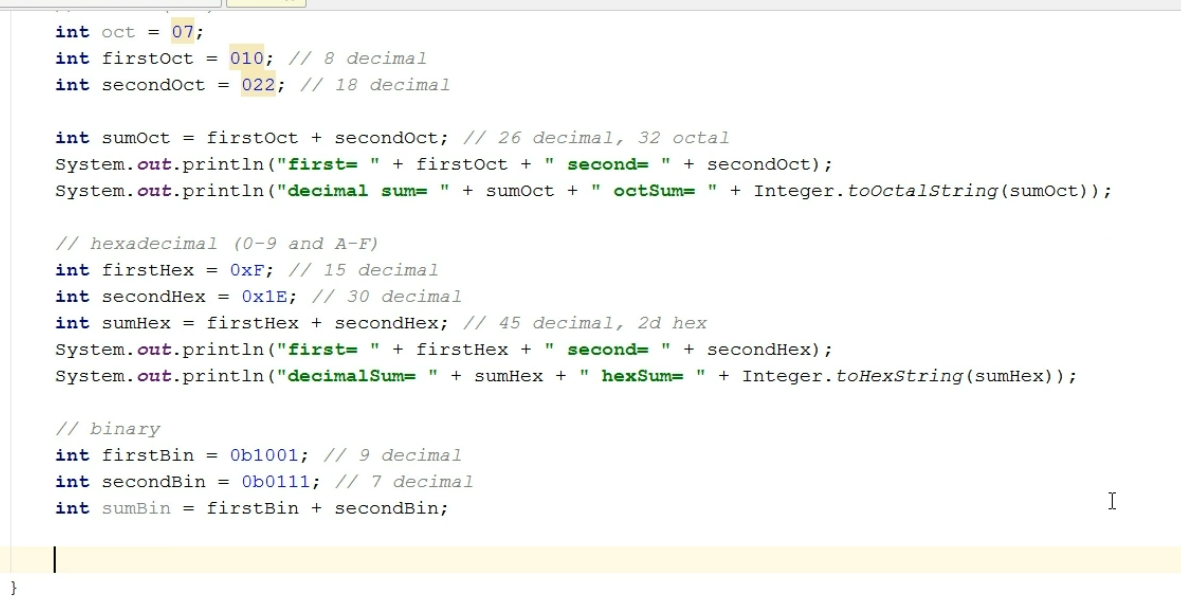
An 8-bit signed integer, also known as a byte in Java, can store values in the range from -128 to 127. The most significant bit (MSB) is reserved for the sign (0 for positive, 1 for negative), and the remaining 7 bits are used to represent the magnitude of the value.



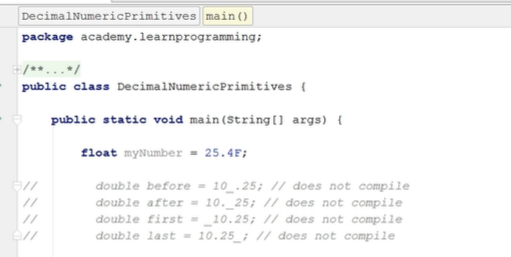
# Assigning literals to variables

In java number literals are by defaults read as integers. So to assign a value to any other data type or to represent them in other formats we need to use appropriate letters.  
  
For example :

* L or l for long
* F or f for floating point number  
    
    
  

**Representation of octal, hexadecimal and binary number**  
  


* + Octal are represented with adding a 0 at the beginning
  + Hexadecimal are represented with adding a 0x at the beginning
  + Binary are represented with adding a 0b at the beginning

**Representation of decimal numbers**  
  
  
  
  
  

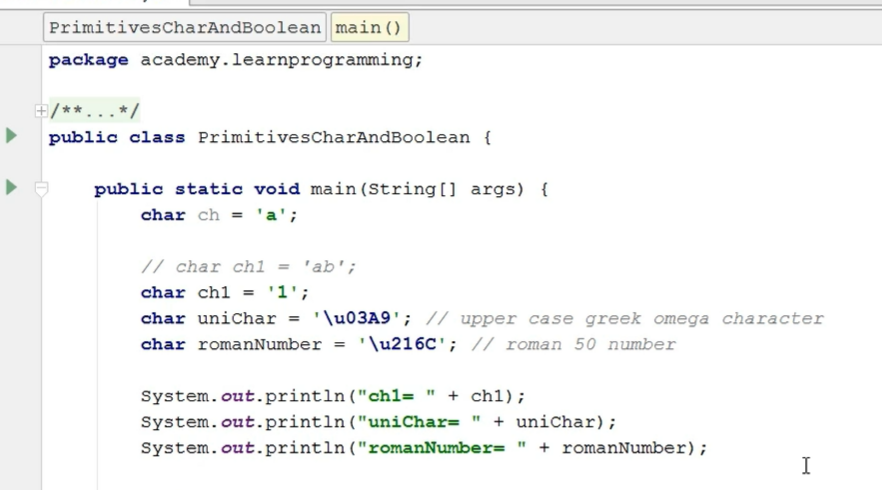

**Hexadecimal**   
The value 0x1.91ebfp1 is a hexadecimal floating-point literal in Java. Java allows you to write floating-point numbers in hexadecimal notation for precise representation of binary fractions.

Here's a breakdown of the hexadecimal floating-point literal 0x1.91ebfp1:

* 0x: Indicates that the number is in hexadecimal format.
* 1.91ebf: The hexadecimal digits of the number.
  + 1: The integral part.
  + .91ebf: The fractional part.
* p1: The exponent part, which is in base 2 (binary). The p is similar to e in decimal scientific notation, but it signifies a power of 2.

**Decimal**for simple base 10 number we use e or E followed by the exponent power as shown in the figure.

## Primitive char and Unicode characters



# Default Values Assigned to Primitive Data Types

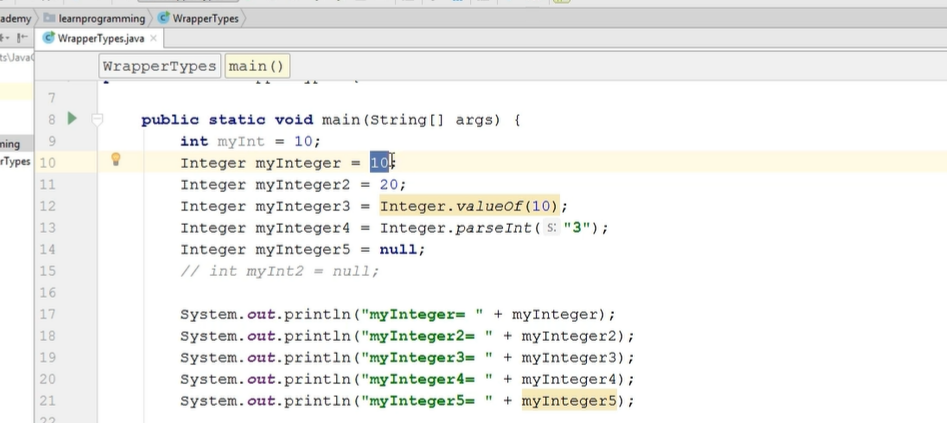
In Java, when a variable is declared but not initialized, it is assigned a default value based on its data type. The default values for the primitive data types in Java are as follows:

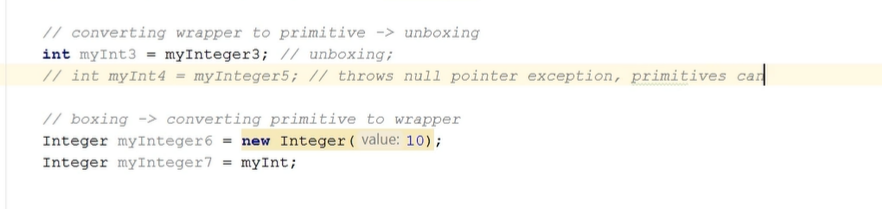
byte: 0  
short: 0  
int: 0  
long: 0L  
float: 0.0f  
double: 0.0d  
char: ‘\u0000’ (null character)  
boolean: false

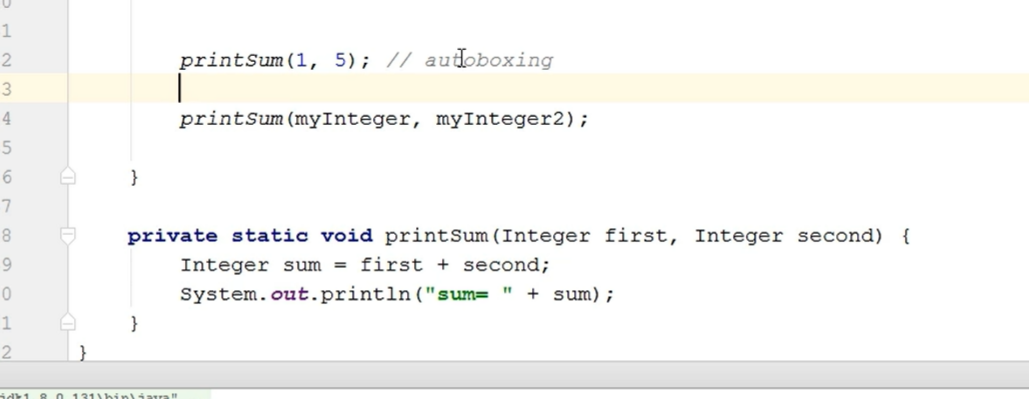
It is important to note that these default values are only assigned if the variable is not explicitly initialized with a value. If a variable is initialized with a value, that value will be used instead of the default.

# It only provides this default value initialization for class variables not for local variables

# Wrapper Classes

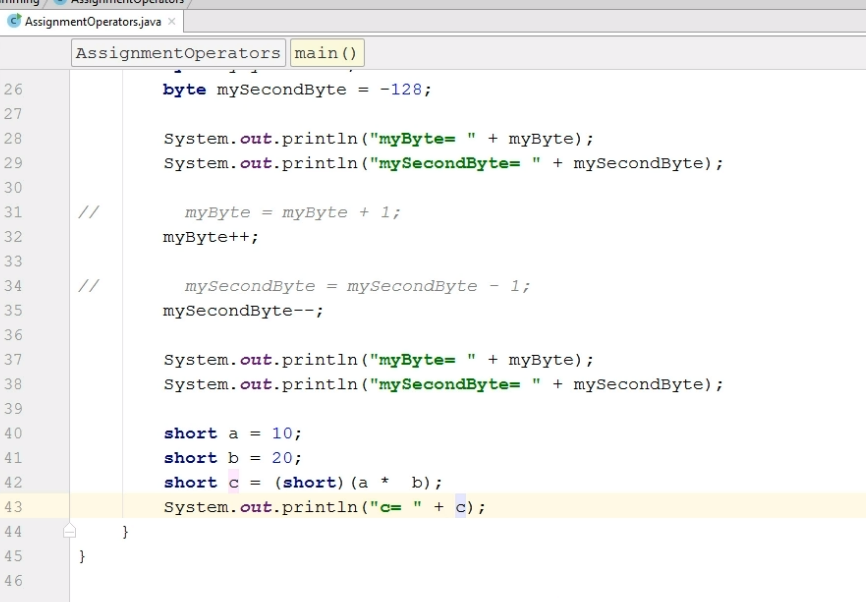


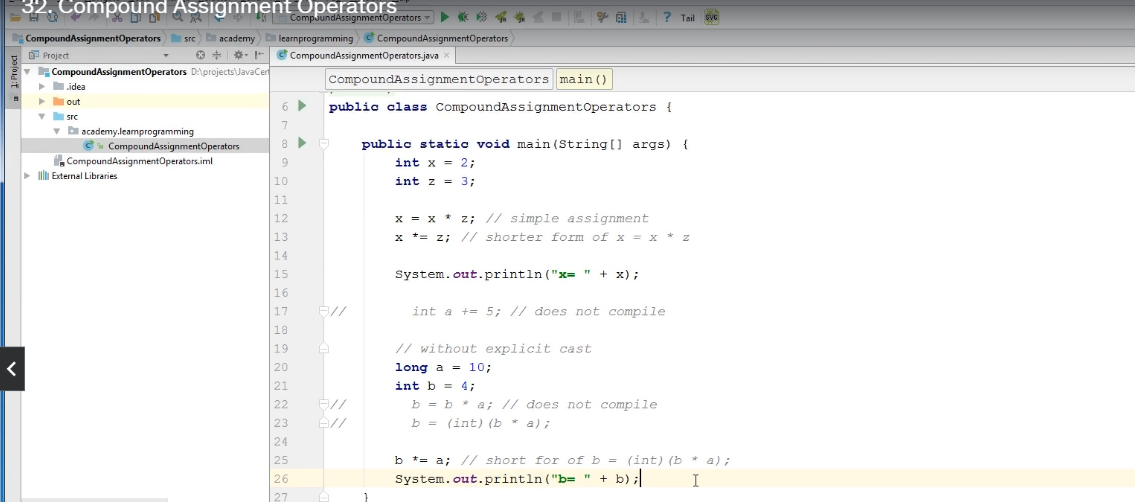


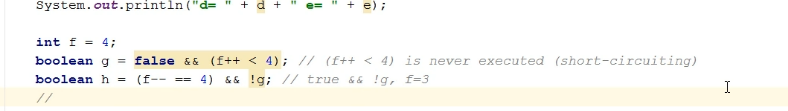
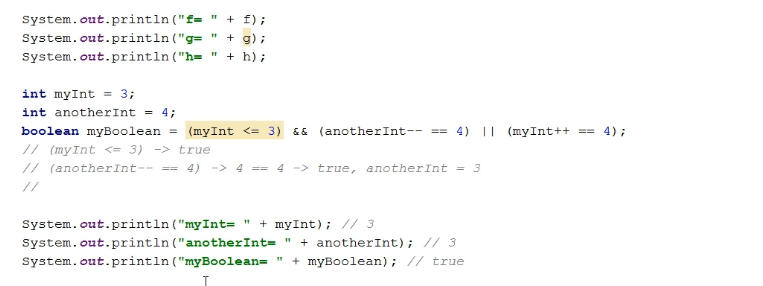


## Boxing and Autoboxing

# Assignment operator





Short circuiting  
  
  
  


anotherInt will remain 3 because of short circuiting

# == and equals method

Int x = 4;

Int y = 4;

Boolean c = x==y; //c = true

Integer x = 4;

Integer y = 4;

Boolean c = x==y // c = true;

Integer x = 128;

Integer y = 128;

Boolean c = x == y // c = false;

Integer x = 4;

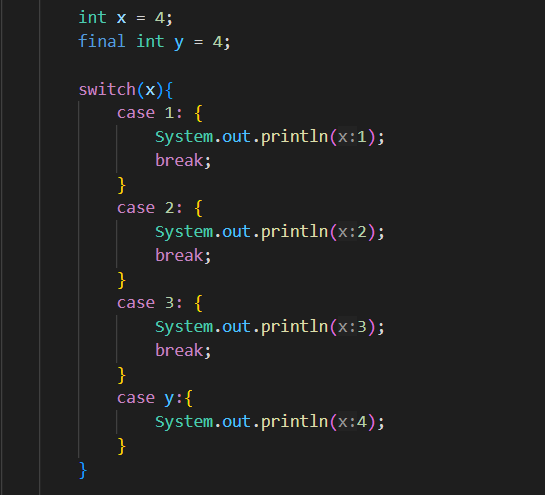
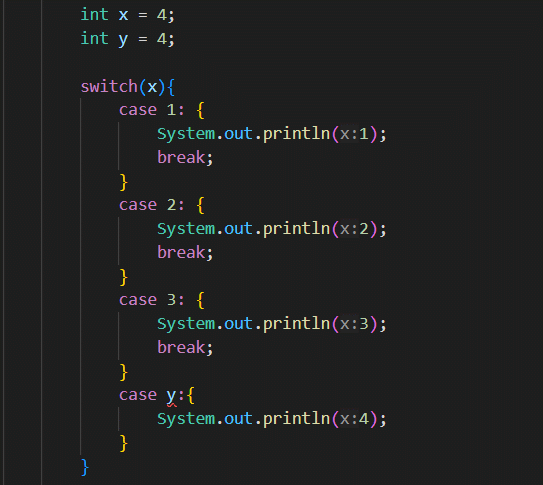
Integer y = new Integer(4);

Boolean c = x==y // c = false;

* The equals operator == when used with primitive data type compares the value stored at the memory location of the variable.
* The equals operator == when used with objects (here of type Integer) then is compares the hash value of the memory location for the object.
* Now it returns true in case of Integer object value from -128 to 127 because there is a cache in Integer wrapper class that gives same memory location to each object made for the same numerical value in this range.
* For numeric values outside this range it creates a new instance of object.
* But if you create an object with the new keyword then no matter the numeric value it creates a new instance of the object.

# The switch case conflict

The Java language specification does not allow non-constant variables in switch case labels. This is because the case labels must be compile-time constants, and non-final variables are not considered compile-time constants.

Hence: while:   


The above code shows compilation error This code above doesn’t   
  
this happens because in the first one y is just a normal integer variable and in the second one it is declared as final integer variable.

# Static initialization blocks

Static initialization blocks in Java are used to initialize static variables or perform other initialization tasks for a class. They are executed when the class is loaded into the Java Virtual Machine (JVM) and only once, regardless of how many instances of the class are created.

Here's the general syntax of a static initialization block:

static { // Initialization code }

The static initialization block is defined using the **static** keyword followed by a pair of curly braces. Inside the block, you can write any valid Java code, including variable assignments, method calls, or other initialization tasks.

Key points to understand about static initialization blocks:

1. Execution Timing: The static initialization block is executed when the class is first loaded into the JVM, before any static variables or static methods of the class are accessed or invoked.
2. Order of Execution: If a class has multiple static initialization blocks, they are executed in the order they appear in the code.
3. Access to Static Members: Inside the static initialization block, you can access and modify static variables and invoke static methods of the class freely. However, you cannot access instance variables or instance methods directly since they are associated with specific object instances.
4. Exception Handling: If an exception occurs during the execution of the static initialization block, the class fails to initialize, and an exception is thrown. This exception can be caught and handled using standard Java exception handling mechanisms.

Static initialization blocks are useful when you need to perform complex initialization logic for static variables or execute additional tasks during class loading. They are commonly used to initialize static variables based on calculations or external data sources, set up static data structures, or establish connections to external resources.

Here's a simple example demonstrating the usage of a static initialization block:

public class MyClass {  
 static int *myStaticVariable*;  
  
 static {  
 // Perform initialization  
 *myStaticVariable* = 42;  
 System.*out*.println("Static initialization block executed");  
 }  
  
 public static void main(String[] args) {  
 System.*out*.println("Value of myStaticVariable: " + *myStaticVariable*);  
 }  
}

In this example, the static initialization block sets the value of **myStaticVariable** to 42. When the class **MyClass** is loaded, the static initialization block is executed, and the message "Static initialization block executed" is printed. Later, in the **main** method, the value of **myStaticVariable** is printed, showing the initialized value.

Static initialization blocks are a powerful feature in Java that allow you to perform necessary setup tasks when a class is loaded. They provide flexibility and help ensure proper initialization of static variables and resources.

To put constraints on the input while using the static initialization block, you can add conditional statements within the block to validate the input and handle any invalid inputs accordingly.

Here's an example:

import java.util.Scanner;  
  
public class MyClass {  
 static int *myStaticVariable*;  
  
 static {  
 Scanner scanner = new Scanner(System.*in*);  
  
 // Read input until a valid value is provided  
 while (true) {  
 System.*out*.print("Enter a positive integer: ");  
 if (scanner.hasNextInt()) {  
 int input = scanner.nextInt();  
 if (input > 0) {  
 *myStaticVariable* = input;  
 break; // Valid input, exit the loop  
 } else {  
 System.*out*.println("Invalid input! Please enter a positive integer.");  
 }  
 } else {  
 System.*out*.println("Invalid input! Please enter an integer.");  
 scanner.next(); // Consume the invalid input  
 }  
 }  
  
 scanner.close();  
 System.*out*.println("Static initialization block executed");  
 }  
  
 public static void main(String[] args) {  
 System.*out*.println("Value of myStaticVariable: " + *myStaticVariable*);  
 }  
}

# Strings

Strings are immutable so any of the below methods will not make any changes to which they are called for instead they will return a new string. (Except the length and char that will return int and char respectively)

**length()**

**charAt()**

**indexOf()**

**Syntax:** indexOf(ch, fromindex) // fromindex parameter is optional

String str = “java is fun”;

str.indexOf(‘a’); // 1

str.indexOf(‘a’, 2); //4

str.indexOf(‘fun’); //8

str.indexOf(‘fun’,10); //-1 match wasn’t found

**substring()**

**Syntax:** substring(beginIndex, endIndex) // endIndex is optional

// beginIndex is inclusive and endIndex is excalusive

// end parameter should always be greater than start parameter.

Str.substring(4,4) // will return empty string

str.substring(4,2) //string index out of range

**toUpperCase()**

**Syntax :** str.toUpperCase();

System.out.println( “abc”.toUpperCase()); // this is a valid syntax

**toLowerCase()**

**equals()**

**Sytax:** str1.equals(str2); // case sensitive

**equalsIgnoreCase()**

ignores case.

**startsWith()**

Syntax: str1.startsWith(str2) || str1.startsWith(“literal”); // case sensitive

**endsWith()**

//case sensitive

**contains()**

//case sensitive

**replace()**

**Syntax :** str1.replace(toReplace, replaceWith);

String myString = “Java is cool”;

System.out.println(myString.replace(‘a’, ‘A’);

**trim()**

**Syntax:** str.trim()

Trims any spaces in the beginning or ending of a string.

System.out.println(“ Java “.trim()); // will remove spaces before and after java

System.out.println(“ Java is cool “.trim()); // will remove spaces before and after whole string Java is cool and not from between.

**Chaining methods**

String str = “ Java “.trim().toLowerCase().replace(‘j’,’J’); // output = Java

// in this first “ Java “.trim() will return “Java” then “Java”.toLowerCase() will return java and so on

## String to int and int to String

* The Integer.parseInt(s) method is a built-in Java method that parses a String representation of an integer and converts it into an int value. It takes a String parameter s and returns an int value that represents the parsed integer.
* To convert an int to a String in Java, you can use the String.valueOf() method or concatenate the int value with an empty String.
* Concatenating with an empty String

Concatenating with an empty String:

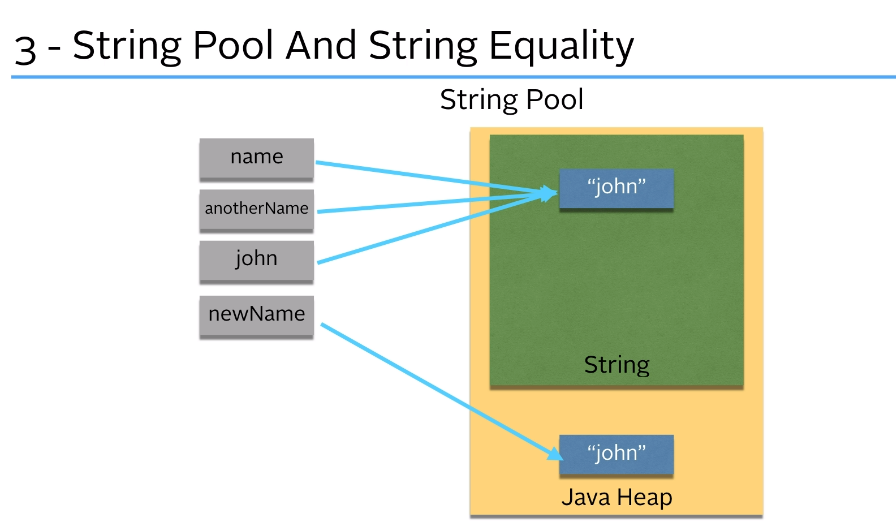
int number = 42;  
String numberString = "" + number;

## Integer.parseInt()

Converts string to primitive int  
  
Integer.valueOf()

Converts string to wrapper class Integer

## String pool and string Equality



When we create a string in java using the String class then it checks if the same string is available in the string pool and if it is then the new string just points to that same string in the java string pool   
But if we use the new keyword to create a string then in that case java creates a new instance of the string in the java heap.  
  
so a string

String name = “John”

String name2 = “John”

String name3 = new String(“John”)

Boolean isEqual;

isEqual = name == name2 // true;

isEqual = name == name3 // false   
  
  
but if we use the equals() method then it will return true in both cases as it checks the actual value of the string and not the hash value

Constant folding at compile time  
When you use:

String str3 = "ab" + "c";

 Both "ab" and "c" are string literals.

 The Java compiler recognizes this and concatenates them at compile time, resulting in the string literal "abc".

 Since "abc" is a string literal, it is interned and stored in the string pool.

Thus the code essentially becomes:

String str3 = "abc";

Both str1 and str3 refer to the same string literal in the string pool.

Runtime Concatenation  
In contrast when you do:

String str3 = str2 + "c";

str2 is a variable, and "c" is a string literal.

The concatenation happens at runtime, resulting in a new string object that is not automatically interned.

Hence

String str1 = "abc"; // String literal "abc", stored in the string pool  
String str2 = "ab"; // String literal "ab", stored in the string pool  
String str3 = str2 + "c"; // Concatenation happens at runtime, creating a new string object  
  
boolean isEqual = str1 == str3; // This will be false

# StringBuilder

 **Purpose**:

* StringBuilder is used to create mutable (modifiable) strings.
* It is more efficient than String when performing multiple modifications, like appending, inserting, or deleting characters.

 **Key Characteristics**:

* **Mutable**: Unlike String, which creates a new object every time it's modified, StringBuilder modifies the object itself.
* **Not thread-safe**: It is not synchronized, so it's not safe for use in multi-threaded environments (use StringBuffer for thread safety).

 **Common Methods**:

* charAt(), indexOf(), length(), substring()
* append(String str): Adds the given string to the end.
* insert(int offset, String str): Inserts the string at the specified position.
* delete(int start, int end): Removes characters from the start index until before the end index. End index is excluded.
* deleteCharAt(int index): delete the character at the given index.
* reverse(): Reverses the sequence of characters.
* toString(): Converts the StringBuilder content to a String.

 **Performance**:

* More efficient than String for concatenation because it doesn't create new objects during modification.
* Ideal for scenarios where many string manipulations are required.

 **Example Usage**:

java

Copy code

StringBuilder sb = new StringBuilder("Hello");

sb.append(" World"); // Appends " World"

sb.insert(6, "Java "); // Inserts "Java " at index 6

sb.reverse(); // Reverses the string

String result = sb.toString(); // Converts to String

 **Constructors**:

* StringBuilder(): Creates an empty StringBuilder.
* StringBuilder(String str): Creates a StringBuilder with the specified string.
* StringBuilder(int capacity): Creates a StringBuilder with the specified initial capacity.

 **Capacity**:

* The capacity of StringBuilder grows automatically as needed, but you can specify it manually for performance optimization.

**Capacity vs. Length in** StringBuilder

* **Length**:
  + Refers to the number of characters currently stored in the StringBuilder.
  + It is the size of the actual content.
  + Use length() method to get the length.

Example:

java

Copy code

StringBuilder sb = new StringBuilder("Hello");

System.out.println(sb.length()); // Output: 5

* **Capacity**:
  + Refers to the total allocated space for characters in the StringBuilder.
  + This is the maximum number of characters it can hold before needing to resize.
  + Use capacity() method to get the capacity.
  + The default capacity is 16 characters unless specified.

Example:

java

Copy code

StringBuilder sb = new StringBuilder("Hello");

System.out.println(sb.capacity()); // Output: 21 (5 + 16 extra capacity)

* **Capacity Resizing**:
  + When the number of characters exceeds the current capacity, StringBuilder automatically increases the capacity (usually doubling it).
  + You can also set the capacity manually using the constructor or ensureCapacity() method.

### **Ways to Specify Capacity:**

1. **Using Constructor**:
   * You can specify the initial capacity of a StringBuilder when creating it.

java

Copy code

StringBuilder sb = new StringBuilder(50); // Initializes with a capacity of 50

1. **Using ensureCapacity(int minimumCapacity)**:
   * You can use the ensureCapacity method to ensure the StringBuilder has at least the specified capacity.

java

Copy code

StringBuilder sb = new StringBuilder();

sb.ensureCapacity(100); // Ensures capacity is at least 100

If you initialize a StringBuilder with the default capacity of 16 and keep appending characters, once you append the 17th character, the StringBuilder will automatically increase its capacity.

### **What Happens Internally:**

1. The initial capacity of a StringBuilder is 16 (if no capacity is specified).
2. When you append characters and the total length exceeds the current capacity (16 in this case), the StringBuilder automatically increases its capacity.
3. The new capacity is typically **(current capacity \* 2) + 2**. So, after reaching 16 characters, the capacity will grow to 34.

## Why Strings are immutable in java?

Strings in Java are immutable for several important reasons, primarily focused on security, performance, and memory efficiency. Here's a breakdown of why Java makes strings immutable:

### 1. **Security**:

* **Immutable strings prevent data tampering**: In Java, strings are commonly used for sensitive data, such as usernames, passwords, network connections, and file paths. If strings were mutable, a malicious actor could modify the string's content, leading to security vulnerabilities.
* **Thread safety**: Since immutable objects cannot be modified after creation, they are inherently thread-safe. Multiple threads can use the same string without needing synchronization, avoiding race conditions or other concurrency issues.

### 2. **Caching and Performance (String Pool)**:

* **String interning**: Java maintains a **string pool**, where strings with the same value are stored only once. This is possible because strings are immutable. If strings were mutable, changing one string's value could unintentionally modify the same string in another part of the program, breaking the pooling mechanism.
* **Memory efficiency**: By storing only one instance of each string literal in the string pool, Java reduces memory consumption, as identical strings can share the same memory.

### 3. **Hashcode Consistency**:

* **Immutable strings ensure consistent hash codes**: In Java, strings are often used as keys in collections like HashMap and HashSet. The immutability of strings guarantees that once a string is created, its hash code will not change, ensuring it remains correctly located in hash-based data structures. If strings were mutable, their hash codes could change, making it impossible to retrieve values stored using the original hash code.

### 4. **Design Simplicity**:

* **Immutable objects are simpler to understand and use**: With immutability, you don't have to worry about unintended changes to a string. Once a string is created, it stays the same throughout the program, making debugging and reasoning about code easier.

### 5. **Security in Class Loading and Reflection**:

* Java uses strings for loading classes and for reflection operations. If strings were mutable, someone could tamper with the names of classes or methods, leading to unpredictable behavior or security

# Date and time

There are three date and time class

* LocalDate
* LocalTime
* LocalDateTime

Date classes are immutable.

## Methods

**now() :** LocalDate.now(), LocalTime.now(), LocalDateTime.now()

# BigDecimal

The **BigDecimal** class in Java is a part of the **java.math** package and provides arbitrary-precision decimal arithmetic. It is used for precise numerical calculations where accuracy is critical, especially when dealing with large numbers or when the precision of floating-point numbers is insufficient.

Using **BigDecimal** can help avoid common pitfalls and inaccuracies associated with floating-point arithmetic. It is particularly useful in financial calculations, currency conversions, and other scenarios where precision is essential.

Here's an example of using **BigDecimal** to perform precise arithmetic calculations:

import java.math.BigDecimal;  
  
public class BigDecimalExample {  
 public static void main(String[] args) {  
 BigDecimal num1 = new BigDecimal("10.5");  
 BigDecimal num2 = new BigDecimal("3.2");  
  
 // Addition  
 BigDecimal sum = num1.add(num2);  
 System.*out*.println("Sum: " + sum); // Output: 13.7  
  
 // Subtraction  
 BigDecimal difference = num1.subtract(num2);  
 System.*out*.println("Difference: " + difference); // Output: 7.3  
  
 // Multiplication  
 BigDecimal product = num1.multiply(num2);  
 System.*out*.println("Product: " + product); // Output: 33.6  
  
 // Division  
 BigDecimal quotient = num1.divide(num2, 2, BigDecimal.ROUND\_HALF\_UP);  
 System.*out*.println("Quotient: " + quotient); // Output: 3.28  
 }

In the line **BigDecimal quotient = num1.divide(num2, 2, BigDecimal.ROUND\_HALF\_UP);**, the **divide** method of the **BigDecimal** class is used to perform division between two **BigDecimal** objects (**num1** and **num2**).

The **divide** method takes three arguments:

1. Divisor (**num2**): This is the **BigDecimal** object that will be divided by the dividend (**num1**).
2. Scale: The scale determines the number of digits to the right of the decimal point in the result. In this case, **2** is passed as the scale, indicating that the result should have two decimal places.
3. Rounding Mode: The rounding mode specifies how the result should be rounded in case there is a need for rounding. In this case, **BigDecimal.ROUND\_HALF\_UP** is used as the rounding mode. It is one of the rounding modes provided by the **BigDecimal** class and rounds towards the nearest neighbor. If the number is exactly halfway between two neighbors, it rounds up.

The **divide** method then performs the division operation and returns a new **BigDecimal** object representing the quotient. This object is assigned to the variable **quotient**.

Here's an example to illustrate the use of **divide** with **scale** and **roundingMode**:

To divide BigDecimal by an integer…

BigDecimal percentage = y.divide(BigDecimal.valueOf(100));

# BigInteger

**BigInteger** is a class in the **java.math** package in Java. It is used for representing and performing arithmetic operations on integers with arbitrary precision. Unlike the primitive data types (**int**, **long**, etc.), **BigInteger** can handle integer values of any size, limited only by the amount of memory available.

Here are some key features and functionalities of **BigInteger**:

1. Arbitrary Precision: **BigInteger** can represent integers of any size, limited only by available memory. This makes it suitable for applications that require operations on extremely large numbers.
2. Immutable: **BigInteger** objects are immutable, meaning their values cannot be changed after they are created. Operations on **BigInteger** return new **BigInteger** objects rather than modifying the original object.
3. Arithmetic Operations: **BigInteger** provides methods for performing arithmetic operations such as addition, subtraction, multiplication, division, and modular arithmetic (remainder and modular exponentiation). These operations can be performed on **BigInteger** objects of different sizes.
4. Bitwise Operations: **BigInteger** supports bitwise operations like bitwise AND, bitwise OR, bitwise XOR, and bitwise shift operations.
5. Comparison and Equality: **BigInteger** provides methods for comparing **BigInteger** objects and checking for equality (**equals**) or inequality (**compareTo**).
6. Conversion: **BigInteger** can be converted to and from other primitive data types like **int**, **long**, **byte**, **short**, and vice versa.

To use **BigInteger**, you need to import the **java.math.BigInteger** class in your Java program. Here's an example of creating and performing arithmetic operations with **BigInteger**:

import java.math.BigInteger;  
  
public class BigIntegerExample {  
 public static void main(String[] args) {  
 BigInteger num1 = new BigInteger("12345678901234567890");  
 BigInteger num2 = new BigInteger("98765432109876543210");  
  
 BigInteger sum = num1.add(num2);  
 BigInteger difference = num1.subtract(num2);  
 BigInteger product = num1.multiply(num2);  
 BigInteger quotient = num1.divide(num2);  
  
 System.*out*.println("Sum: " + sum);  
 System.*out*.println("Difference: " + difference);  
 System.*out*.println("Product: " + product);  
 System.*out*.println("Quotient: " + quotient);  
 }  
}



# To check if all digits of a number are pairwise different

The statement **String.valueOf(x)** converts the integer **x** to its corresponding string representation. It is used to convert the number **x** into a string so that we can apply string operations on it.

In the context of the code snippet, **isDistinct(String.valueOf(x))** is used to check if each digit of the number **x** is distinct. Here's how it works:

1. **String.valueOf(x)**: This converts the integer **x** to a string representation. For example, if **x** is **12345**, **String.valueOf(x)** will return the string **"12345"**. This allows us to access individual digits of the number.
2. **isDistinct(String.valueOf(x))**: This calls the **isDistinct** method with the string representation of **x** as the argument. The **isDistinct** method checks if each digit in the string is distinct. If all the digits are distinct, it returns **true**; otherwise, it returns **false**.

So, **isDistinct(String.valueOf(x))** is a way to check if each digit of the number **x** is

# Why we use srand in c++;

|  |  |
| --- | --- |
|  | Seeding the random number generator with the current time is a common practice to make sure that the sequence of random numbers generated is different every time the program runs. The reason for this is that computers generate random numbers using deterministic algorithms, which means that if you start with the same initial conditions (known as the seed), you will get the same sequence of random numbers.  By using the current time as the seed, you introduce an element of unpredictability. Since the time changes every second, the seed will be different every time the program runs, leading to a different sequence of random numbers. |

# Abstract method

ABSTRACT CLASS

A class having atleast one abstract method

ABSTRACT METHOD

A method whose implementation is not defined - such methods are meant to be overridden

When an abstract class is subclassed, the subclass usually provides implementations for all of the methods in parent class. If it doesn’t, it must be declared abstract.

We cannot create object of an abstract class

# Lambda Expression

A lambda expression is, essentially an anonymous(that is unnamed) method. However this method is not executed on its own. Instead it is used to implement a method defined by a functional interface.

Thus, a lambda expression results in a form of anonymous class. Lambda expressions are also commonly referred to as closures.

A functional interface is an interface that contains one and only one abstract method. Normally this methos specifies the intended purpose of the interface.

# Interface

Interface variables are by default final static

# File Handling

import java.io.File;

File obj = new File(“filename.extension”);  
Creates a file object to help manipulate that file

1. Creating a file  
   Use the createNewFile() method

This method returns a boolean value: true if the file was successfully created, and false if the file already exists.

File myFile = new File("file1.txt");  
try {  
 myFile.createNewFile();  
} catch (IOException e) {  
 System.*out*.println("Unable to create file");  
 throw new RuntimeException(e);  
}

to create a file in a specific directory (requires permission), specify the path of the file and use double backslashes to escape the "\" character (for Windows). On Mac and Linux you can just write the path, like: /Users/name/filename.txt

File myObj = new File("C:\\Users\\MyName\\filename.txt");

1. Writing into file

Inn the following example, we use the FileWriter class together with its write() method to write some text to the file we created in the example above. Note that when you are done writing to the file, you should close it with the close() method:

try {  
 FileWriter fw = new FileWriter("file1.txt");  
 fw.write("Hello World\nI am Anannya");  
 fw.close();  
} catch (IOException e) {  
 throw new RuntimeException(e);  
}

3. Reading from a file

we use the Scanner class to read the contents of the text file

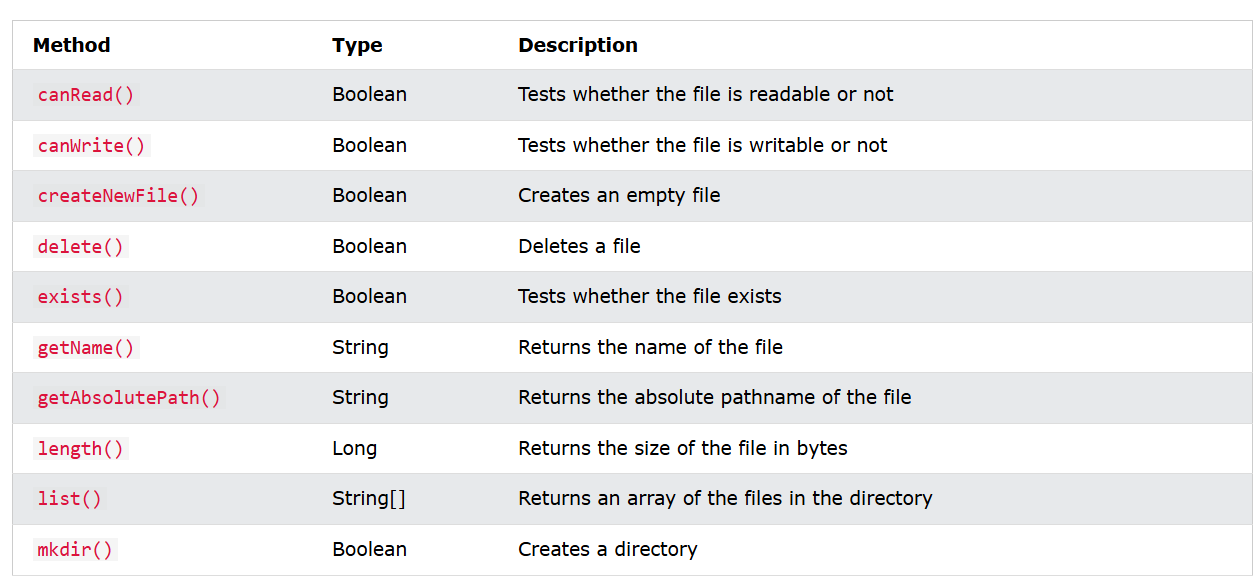
File myFile = new File("file1.txt");  
try {  
 Scanner input = new Scanner(myFile);  
 while (input.hasNextLine()) {  
 String line = input.nextLine();  
 System.*out*.println(line);  
 }  
 input.close();  
} catch (FileNotFoundException e) {  
 throw new RuntimeException(e);  
}

1. Deleting a file

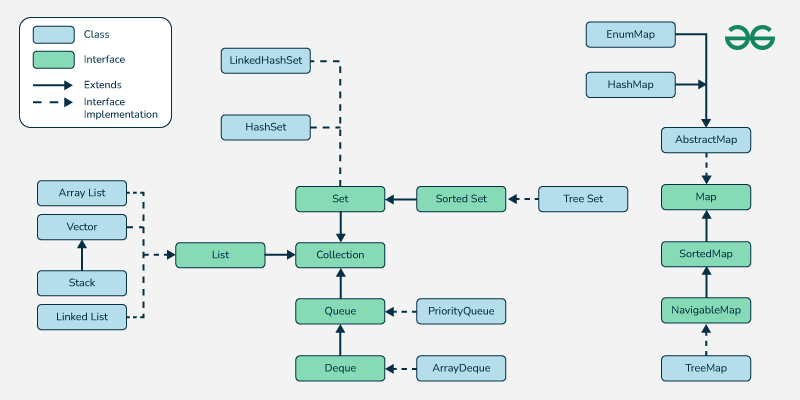
File myFile = new File("file1.txt");  
if(myFile.delete()){  
 System.*out*.println("File deleted successfully "+ myFile.getName());  
}  
else{  
 System.*out*.println("File could not be deleted "+ myFile.getName());  
}

You can also delete a folder. However, it must be empty:

1. Other methods in java



# Collection Framework



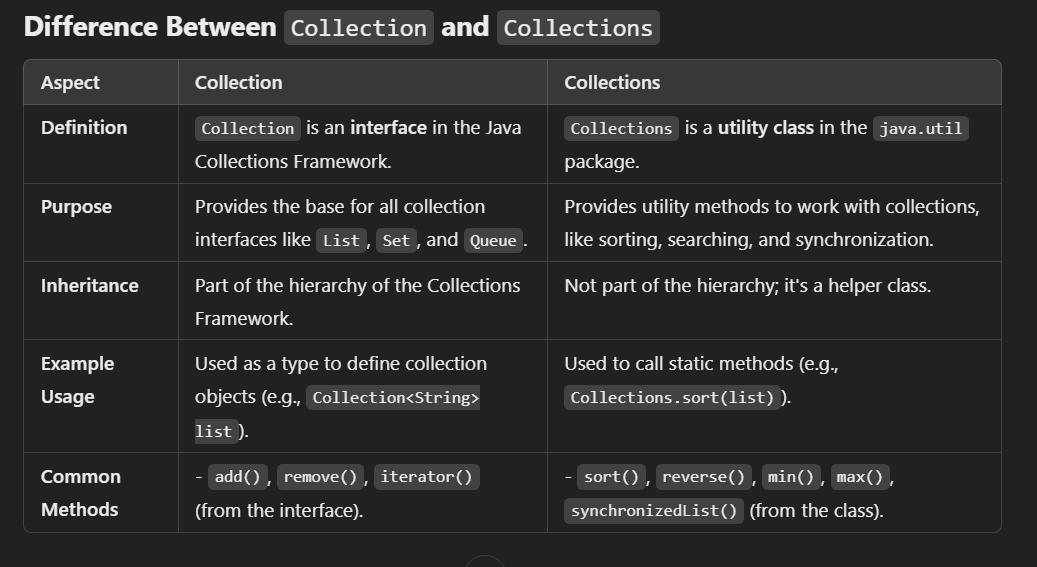
### **1. [Iterable Interface](https://www.geeksforgeeks.org/iterable-interface-in-java/)**

This is the root interface for the entire collection framework. The collection interface extends the iterable interface. Therefore, inherently, all the interfaces and classes implement this interface. The main functionality of this interface is to provide an iterator for the collections. Therefore, this interface contains only one abstract method which is the iterator. It returns the

Iterator iterator();

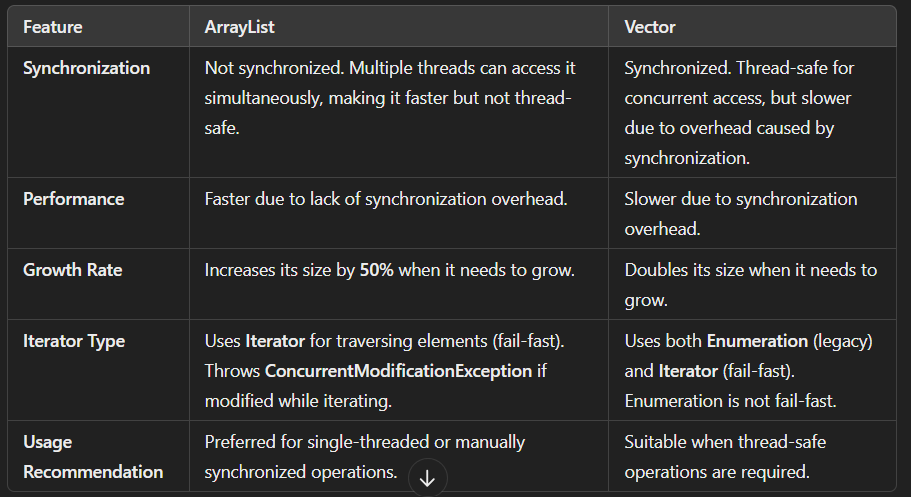
### **2.**[**Collection Interface**](https://www.geeksforgeeks.org/collection-interface-in-java-with-examples/)

This interface extends the iterable interface and is implemented by all the classes in the collection framework. This interface contains all the basic methods which every collection has like adding the data into the collection, removing the data, clearing the data, etc. All these methods are implemented in this interface because these methods are implemented by all the classes irrespective of their style of implementation. And also, having these methods in this interface ensures that the names of the methods are universal for all the collections. Therefore, in short, we can say that this interface builds a foundation on which the collection classes are implemented.



## Difference between arraylist and vectors

1. ArrayList are not thread safe …vectors are thread safe



Queue

Queue<Integer> q = new LinkedList<>();

Being an interface the queue needs a concrete class for the declaration and the most common classes are the [PriorityQueue](https://www.geeksforgeeks.org/priority-queue-class-in-java-2/) and [LinkedList](https://www.geeksforgeeks.org/linked-list-in-java/) in Java. Note that neither of these implementations is thread-safe. [PriorityBlockingQueue](https://www.geeksforgeeks.org/priorityblockingqueue-class-in-java/) is one alternative implementation if the thread-safe implementation is needed.

Stack

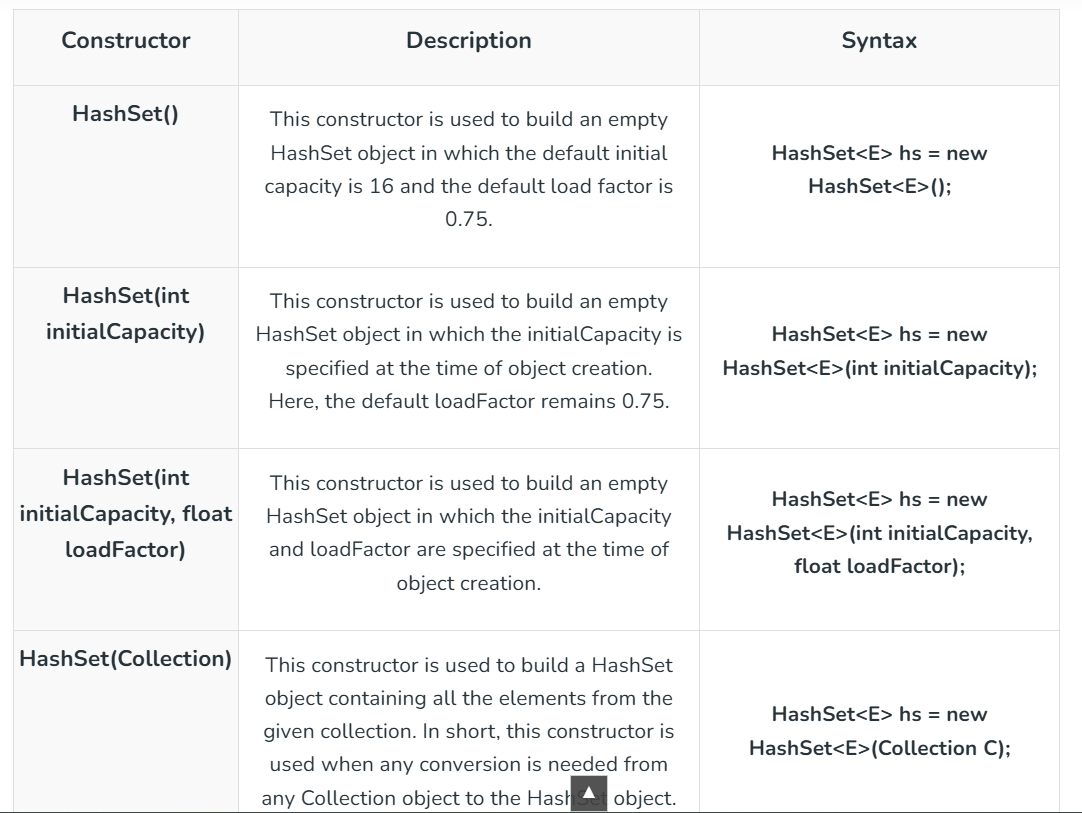
Stack<Integer> stack = new Stack<>();

* The Stack class extends Vector and provides additional functionality specifically for stack operations, such as **push, pop, peek, empty,**and **search**.
* The Stack class can indeed be referred to as a subclass of Vector, inheriting its methods and properties.

# Set in Java

* **No Specific Order**: Does not maintain any specific order of elements (Exceptions: LinkedHashSet and TreeSet).
* **Allows One Null Element**: Most Set implementations allow a single null element.
* **Implementation Classes**: [HashSet](https://www.geeksforgeeks.org/hashset-in-java/) , [LinkedHashSet](https://www.geeksforgeeks.org/linkedhashset-in-java-with-examples/" \t "_blank) and [TreeSet](https://www.geeksforgeeks.org/treeset-in-java-with-examples/" \t "_blank).
* **Thread-Safe Alternatives**: For thread-safe operations, use [ConcurrentSkipListSet](https://www.geeksforgeeks.org/concurrentskiplistset-in-java-with-examples/" \t "_blank)or wrap a set using Collections.synchronizedSet().

HashSet uses HashMap for its internal implementation.  
But HashMap stores key-value pair and HashSet only has values.  
so the elements of the HashSet works as Key in the HashMap and java uses a constant variable for the value part.  
So in a HashMap implementing HashSet all values are same.



The **load factor** in a HashSet (and its underlying implementation, HashMap) is a measure of how full the hash table is allowed to get before its capacity is automatically increased (rehashing occurs). It determines the balance between time complexity and space utilization.

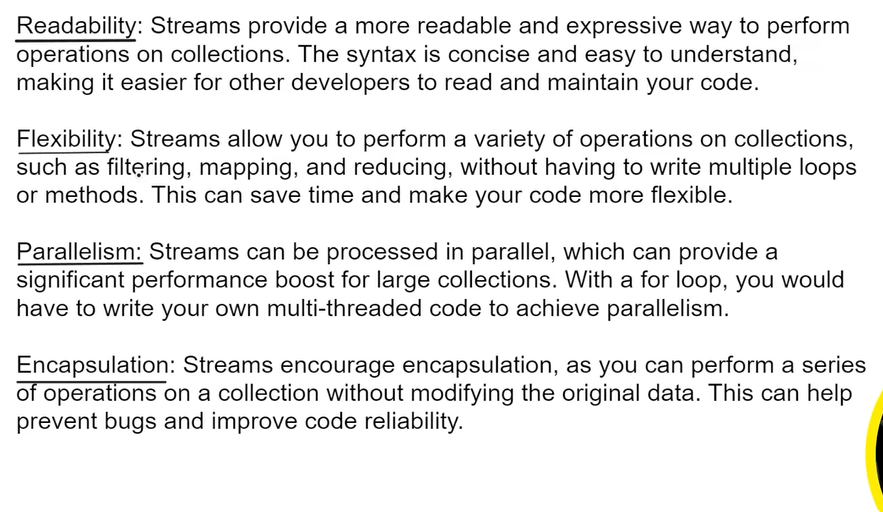
# Map

Maps are not synchronized by default so to use them in multithread environment we can wrap the map using Collections.SynchronizedMap() method.

*Map<K, V> synchronizedMap = Collections.synchronizedMap(new LinkedHashMap<>());*

# Streams

Advantages of using Stream



Creating a Stream

List<String> list = Arrays.*asList*("apple", "banana", "mango");  
Stream<String> stream = list.stream();  
System.*out*.println(stream);  
  
String[] array = {"apple", "banana", "mango"};  
Stream<String> stream2 = Arrays.*stream*(array);  
System.*out*.println(stream2);  
  
Stream<Integer> stream3 = Stream.*of*(1, 2, 3);  
System.*out*.println(stream3);  
  
Stream<Integer> stream4 = Stream.*iterate*(0, n -> n + 1).limit(100);  
System.*out*.println(stream4);

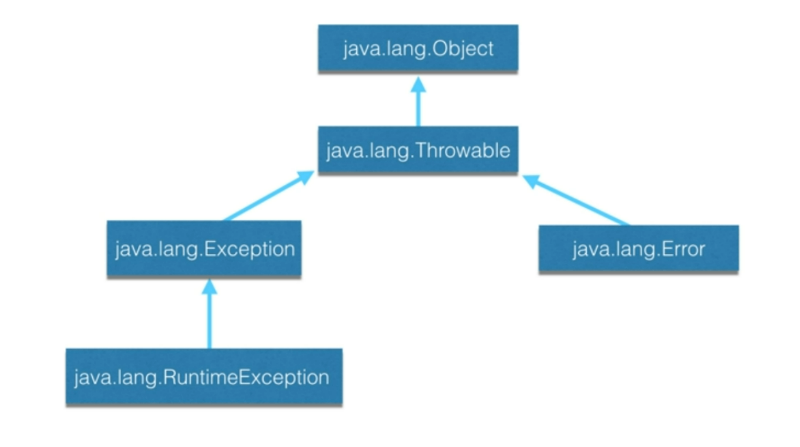
Once you have used a stream you can not reuse it.

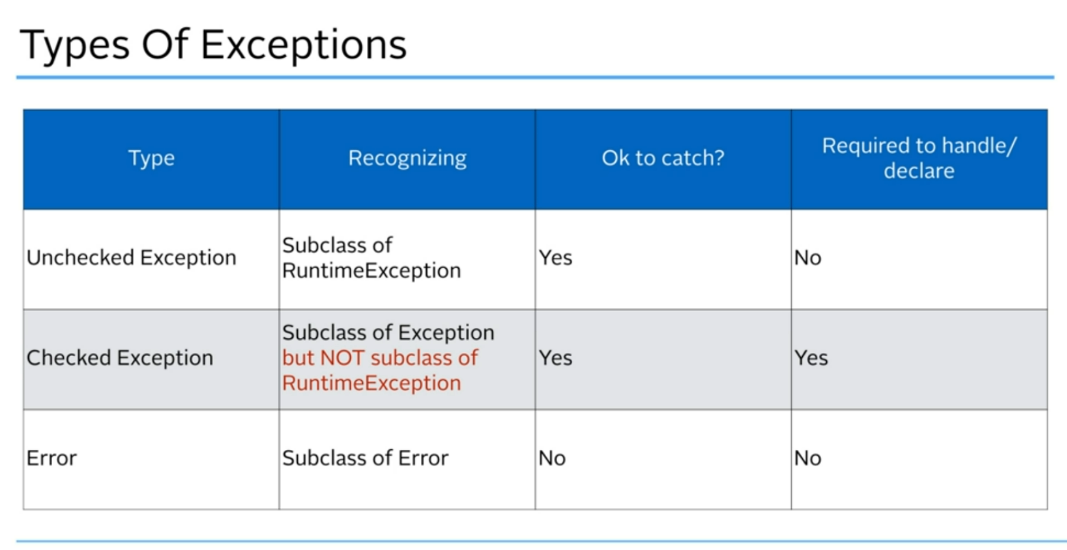
Stream<Integer> stream4 = Stream.*iterate*(0, n -> n + 1).limit(10);  
stream4.forEach(n -> System.*out*.print(n+", "));  
  
System.*out*.println();  
stream4.forEach(n -> System.*out*.print(n+", "));  
//this will give runtime exception  
//Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed

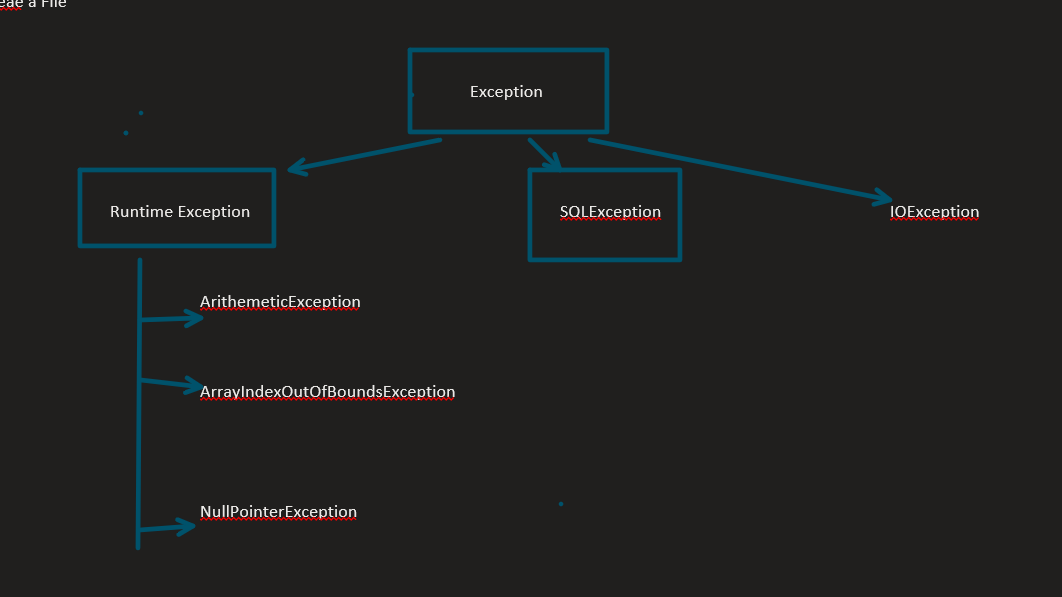
System.*out*.println();  
List<Integer> nums = Arrays.*asList*(1, 2, 3, 4);  
Stream<Integer> numsStream = nums.stream();  
Stream<Integer> doubled = numsStream.map(n-> n\*2);  
doubled.forEach(n->System.*out*.print(n+", "));  
  
//Writing the same in a compact way  
System.*out*.println();  
nums.stream().map(n->n\*2).forEach(n->System.*out*.print(n+", "));  
//this will not return any stream as the forEach loop has already used it  
/\*  
Stream<Integer> doubledData = nums.stream().map(n->n\*2).forEach(n->System.out.println(n));  
\*/  
//This code will give an error.

You might think we are creating so many streams so we must be wasting memory but each stream will be closed after being used once. So we are just kind of replacing not filling up the memory.  
  
We are just manipulating data without doing any modifications to the original data here the nums list.

# Exceptions in Java







All runtime exceptions are unchecked expection so it is not required to check these exception  
Expections like SQLExecption, IOException are checked exceptions and must be handled.

This happens when the method we are using throws a particular type of Exception then in that case when using that method we will have to handle that exception.

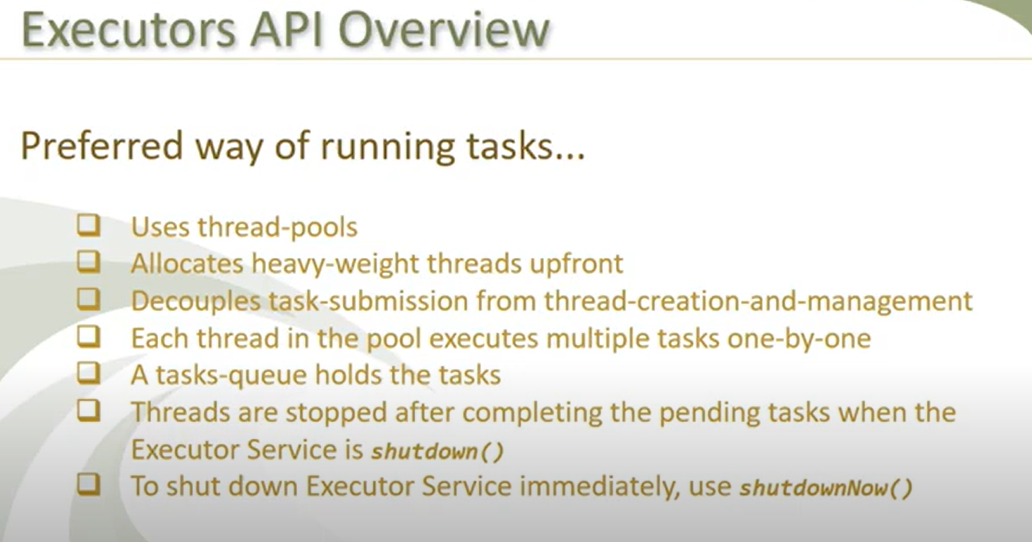
# Executors Framework in java

The[java.util.concurrent.Executors](https://www.geeksforgeeks.org/java-util-concurrent-executor-interface-with-examples/) provide factory methods that are being used to create [ThreadPools](https://www.geeksforgeeks.org/thread-pools-java/) of worker threads. Thread pools overcome this issue by keeping the threads alive and reusing the threads. Any excess tasks flowing in, that the threads in the pool can’t handle are held in a Queue. Once any of the threads get free, they pick up the next task from this queue. This task queue is essentially unbounded for the out-of-box executors provided by the JDK.

To implement this there is an interface called ExecutorService in java that a class will need to extend  
  
There are two built-in implementation of Executors interface  
1. threadPoolExecutor

2. scheduledExecutorService

- This implements the ScheduledExecutorService interface which in turn extends ExecutorService



<https://youtu.be/R8o9AqPxu5s?si=OD_ptq1oDUndB5CI>

**Some types of Java Executors are listed below:**

1. SingleThreadExecutor
2. FixedThreadPool(n)+
3. CachedThreadPool
4. ScheduledExecutor

**Executor 1: SingleThreadExecutor**

A thread pool of single thread can be obtained by calling the static *[newSingleThreadExecutor()](https://www.geeksforgeeks.org/difference-between-executorservice-execute-and-submit-method-in-java/)* method of the Executors class. It is used to execute tasks sequentially.

This can be used when we want to synchronize or serialize access to a common resource.

ExecutorService executor = Executors.*newSingleThreadExecutor*();

**Executor 2: FixedThreadPool(n)**

As the name indicates, it is a thread pool of a fixed number of threads. The tasks submitted to the executor are executed by the n threads and if there is more task they are stored on a LinkedBlockingQueue. It uses Blocking Queue.

ExecutorService fixedPool = Executors.*newFixedThreadPool*(2);

**Executor 3: CachedThreadPool**

Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available. Calls to execute will reuse previously constructed threads if available. If no existing thread is available, a new thread will be created and added to the pool. It uses a [SynchronousQueue](https://www.geeksforgeeks.org/java-program-to-implement-synchronousqueue-api/) queue.

ExecutorService executorService = Executors.*newCachedThreadPool*();

**Executor 4: ScheduledExecutor**

Scheduled executors are based on the interface [ScheduledExecutorService](https://www.geeksforgeeks.org/scheduledexecutorservice-interface-in-java/) which extends the [ExecutorService interface](https://www.geeksforgeeks.org/java-util-concurrent-executorservice-interface-with-examples/). This executor is used when we have a task that needs to be run at regular intervals or if we wish to delay a certain task.

ScheduledExecutorService scheduledExecService = Executors.*newScheduledThreadPool*(1);

* The tasks can be scheduled using either of the two methods:
  + **scheduleAtFixedRate**: Executes the task with a fixed interval, irrespective of when the previous task ended.
  + **scheduleWithFixedDelay**: This will start the delay countdown only after the current task completes.`

# CompletableFuture

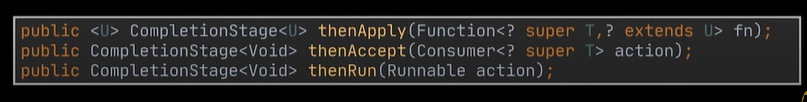
Used for:

* asynchronous programming in java.
* Writing non-blocking code by running a task on a separate thread rather than the main application thread
* The main thread will be able to execute other tasks in parallel
* CompletableFuture implements the Future and the CompletionStage interfaces.
* It provides a huge set of methods for creating, chaining and combining multiple Futures
* It also has a very comprehensive exception handling support.

public class CompletableFuturePrac {  
 public static void main(String[] args) {  
 CompletableFuture<String> completableFuture = CompletableFuture.*supplyAsync*(() ->{  
 System.*out*.println("Started async thread : "+Thread.*currentThread*().getName());  
 try{  
 Thread.*sleep*(5000);  
 }  
 catch(InterruptedException e){  
 e.printStackTrace();  
 }  
 return "OK";  
 });  
  
 System.*out*.println("Main function working on thread : "+Thread.*currentThread*().getName());  
 System.*out*.println(completableFuture.getNow("Hello World"));  
 System.*out*.println("Main function working on thread : "+Thread.*currentThread*().getName());  
 try {  
 System.*out*.println(completableFuture.get());  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 } catch (ExecutionException e) {  
 throw new RuntimeException(e);  
 }  
 }  
}

* The CompletableFuture.get() is blocking as it waits until the Future is completed and then returns the result.

We should be able to attach a callback to the CompletableFuture which should automatically get called when the Future completes. This can be done with the help of the following functions



# Java 8

By java 8 was introduced:-

1. Concise and minimal code.

2. We were missing functional programming in OOPS nature. To utilize functional programming benefits while still being OOP language

3. To enable parallel programming.

**Features of JAVA 8**

1. LAMBDA expression

2. Stream API

3. Date and Time API

4. Base64 Encode and Decode

5. Method reference and constructor reference

6. Default methods in Interface: Before Java 8 interfaces used to only have abstract methods but now they can also have default and static methods.

7. Functional Interface: SAM – single abstract method

- @FunctionalInterface annotation

8. Improvements in JAVA io Collection API and introduction of optional classes

## Lambda Expression

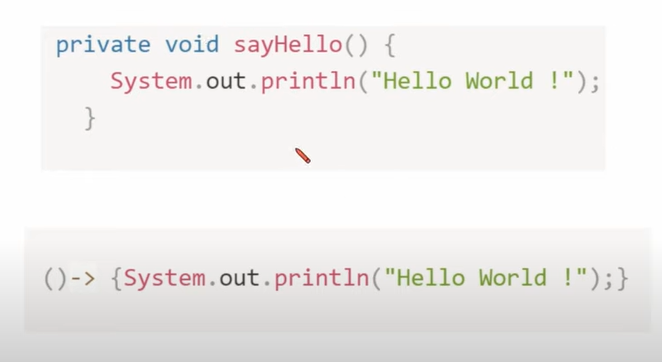
Lambda expression is an anonymous function

That is:-

1. Not having any name

2. Not having any return type

3. Not having modifier

**Steps to make any function lambda expression**

1. Remove modifier

2. Remove return type

3. Remove method name

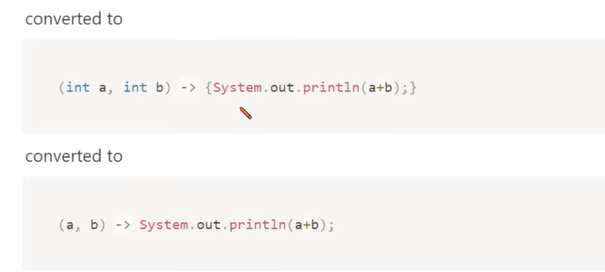
4. Place arrow

**Characteristics / Properties of lambda expression**

* 1. If body has just one statement, then we can remove curly brackets.



* 1. Use type inference, compiler will guess the situation and context.



* 1. No return Keyword



* 1. If only one parameter, then remove parenthesis.



## Functional Interface

Interface having exactly single abstract method but can have any number of defaults and static method. We can invoke lambda expression by using functional Interface.

# Adding the @FunctionalInterface annotation to an interface just helps us identify that the interface is supposed to a functional interface and adding more than one abstract method to that interface will give error. It basically restricts the interface to be functional interface. So if people have already lambda expression and some new team member added another abstract method in the interface all lambda expressions will throw error.

**Inheritance in Functional Interface**

- If there is an interface which has a single abstract method then it is a functional Interface.

- If another interface with empty body or with no abstract methods implements this parent interface then the child will also be a functional interface because it will inherit the abstract method from its parent

- But if you will declare another abstract method in the child interface other than the same abstract method declared in the parent interface then the child will no longer be a functional interface.

- If the child will declare the same abstract method present in parent and no other abstract method then also it will be a functional interface.

- The child can obviously have any number of default and static methods.

## Default methods inside interface

Until 1.7 only public abstract methods were allowed whether we declare by writing or not.  
similarly public static final variables were allowed.  
Since java 8 we can have concrete methods as well inside interface

Defaults methods are nothing but methods with body.

Use default keyword.  
  
Now as we are allowed to have default methods inside interface so we can use these interfaces directly from its child class without having to overriding it.

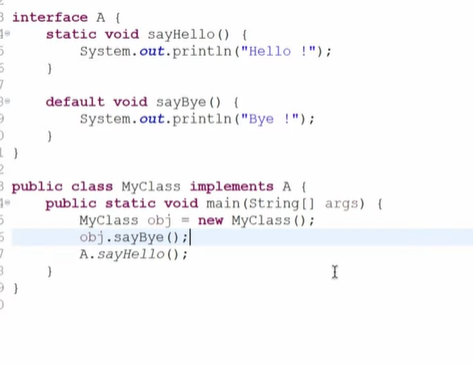
But we can still not make an object of an interface because the interface can still have abstract method.

If a class implements 2 interface and both of them have same default method then there will be an error. We can override it in the child class to remove error. So call one of the methods inside the child class usng super class A.super.sayHello().

## Static methods inside interface

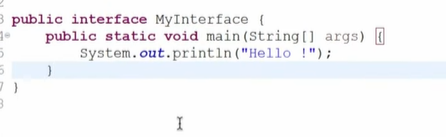
Static methods are one with the keyword static with them and have a body of its own in the interface.  
Static methods can not be overridden or changed in the implementation class.

To call a static method in a class that implements its interface we can not do it with the help of instantiating its object instead use interfacename.staticmethodname()



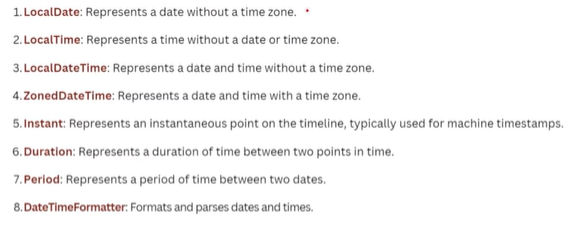
But we can call default method with its obj.

# A class that implements an interface then it can implement a method with the same name. But it is actually not overriding that static method because the child class can not access it anyway. That static method can only be accessed by the name of the interface and not the class name or object. So we are actually creating a new object.

# now as static method is allowed inside interface since Java8 so we can also define the main method inside an interface  


# Date-Time API

Classes in this API



# Java11

## Var in java11

The **var reserved type name (not a Java keyword)** was introduced in Java 10. Type inference is used in var keyword in which it detects automatically the datatype of a variable based on the surrounding context.

Var has some restricted usage

1. We can declare any datatype with the var keyword.

2. var can be used in a local variable declaration.

3. var cannot be used in an instance and global variable declaration.

4. var cannot be used as a Generic type.

5. var cannot be used with the generic type.

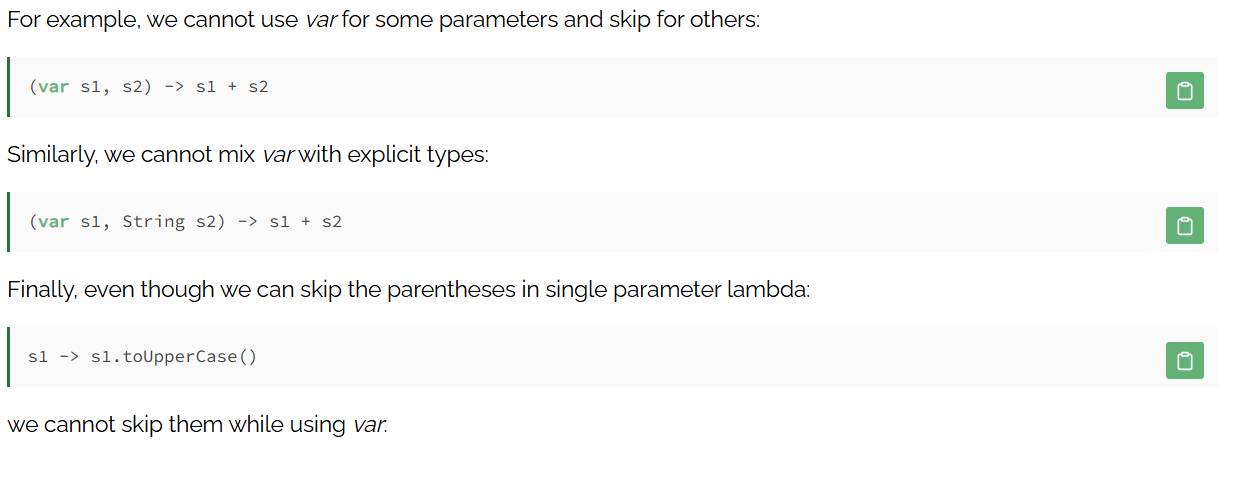
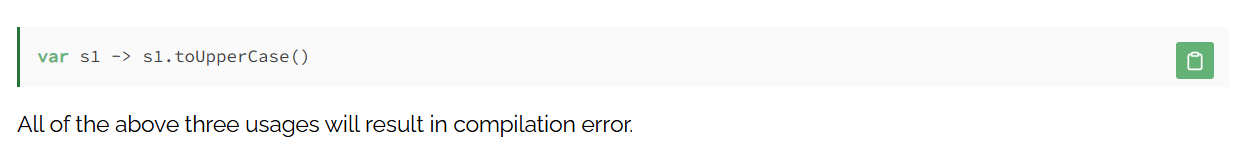
6. var cannot be used without explicit initialization.

7. var cannot be used with Lambda Expression.

8. var cannot be used for method parameters and return type.

Var was introduced in java 10. Java 11 just allowed its usage in Lambda expression

There are again some restrictions with using var in a lambda expression

## Java 11 string methods

- isBlank(): This method is used to check whether the string is blank or not. It returns true if the string is empty or contains only white space codepoints, otherwise false.

- lines() : This method is used to get a stream of lines extracted from the string, separated by line terminators.

- strip(): This method is used to remove all the leading and trailing spaces from a string. If you want to remove only leading spaces then use the stripLeading() method and for trailing, spaces use the stripTrailing() method.

- repeat(): This method is used to repeat a string at the specified time. It returns a string whose value is the concatenation of this string repeated specified times.

# Java 17

## Sealed Classes

Sealed classes provide a mechanism to restrict the inheritance hierarchy for classes and interfaces. This means that you can control which classes can extend or implement a sealed class, thus offering more control over your codebase’s architecture.

public sealed class Shape permits Circle, Square, Triangle {  
 // Class members and methods  
}

**Sealed classes** (introduced in **Java 17**) let you control which classes or interfaces can extend or implement them. This provides:

1. **Better control over inheritance.**
2. **Improved code clarity and safety.**

By using sealed, you explicitly define which classes or interfaces can inherit from the sealed class.

### **Why Use Sealed Classes?**

* **Restrict inheritance:** Prevent unwanted or accidental extensions.
* **Improve maintainability:** Makes it clear which subclasses exist.
* **Control hierarchy:** Works well when the class hierarchy is fixed or predictable.

# Subclasses of a sealed class must explicitly declare themselves as final, sealed or non-sealed.

The requirement for subclasses of a sealed class to explicitly declare themselves as final, sealed or non-sealed exists to enforce controlled inheritance and make the class hierarchy more predictable and well-defined.

* If a subclass is final, you know no further subclassing is possible.
* If it is sealed, the scope of allowed subclasses is again controlled.
* If it is non-sealed, developers are aware that further extension is possible.

**Benefits of sealed classes**

1. Control Inheritance: Only the listed subclasses can extend the sealed class.

2. Prevent Bugs: Avoid unintended extensions or misuse of the base class.

3. Clear documentation: Developers immediately know the complete inheritance hierarchy.

## Instanceof and Pattern Matching

<https://www.geeksforgeeks.org/pattern-matching-for-instanceof-java-17/?ref=asr1>

Pattern Matching in Java 17 enhances the way we work with object types and allows us to write cleaner, safer, and more readable code. It works by allowing type tests and casts to happen in a single operation using the instanceof operator.

**What is Pattern Matching?**

Pattern matching enables you to check the type of an object and, at the same time, bind it to a variable in a single, concise step. This avoids repetitive and verbose code.

**Before Pattern Matching**

In older versions of Java, using instanceof required both a type check and a type cast, leading to redundancy.

#### Example:

java

CopyEdit

Object obj = "Hello, World!";

if (obj instanceof String) {

String str = (String) obj; // Explicit cast

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

}

Here:

1. We check if obj is a String.
2. We cast obj to String manually to use it.

**With Pattern Matching (Java 17)**

With pattern matching, the type test (instanceof) and type cast happen together:

#### Example:

java

CopyEdit

Object obj = "Hello, World!";

if (obj instanceof String str) { // Type test + Cast

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

}

Here:

* obj instanceof String str both checks if obj is a String and casts it to str.
* The variable str is directly accessible within the scope of the if block.

**Benefits of Pattern Matching**

1. **Less Boilerplate:** Eliminates the need for explicit casting.
2. **Cleaner Code:** Reduces redundancy in type checks and casts.
3. **More Readable:** The intent is clear and concise.

**Scope of the Pattern Variable**

The variable declared in instanceof is only accessible within the block where the condition evaluates to true.

#### Example:

java

CopyEdit

Object obj = "Java 17";

if (obj instanceof String str) {

System.out.println("String value: " + str);

} else {

// 'str' is not accessible here

System.out.println("Not a String");

}

**Combining Pattern Matching with Logical Operators**

You can combine pattern matching with logical operators (&&, ||).

#### Example:

java

CopyEdit

Object obj = "Java";

if (obj instanceof String str && str.length() > 3) {

System.out.println("Long String: " + str);

} else {

System.out.println("Not a long String or not a String");

}

## Using pattern matching in switch (Preview feature)

Java 17 also introduces **pattern matching in switch** as a preview feature. It allows switching based on the type of an object.

static void printType(Object obj) {  
 switch (obj) {  
 case String str -> System.*out*.println("String: " + str.toUpperCase());  
 case Integer i -> System.*out*.println("Integer: " + (i \* 2));  
 case null -> System.*out*.println("Null value");  
 default -> System.*out*.println("Unknown type");  
 }  
}  
  
printType("Hello");  
printType(42);  
printType(null);

Here:

* The switch statement directly checks the type of the object and assigns it to the corresponding variable (str, i).
* Each case handles a specific type.

# As this is a preview feature it is not available by default. So to use it compile your code with

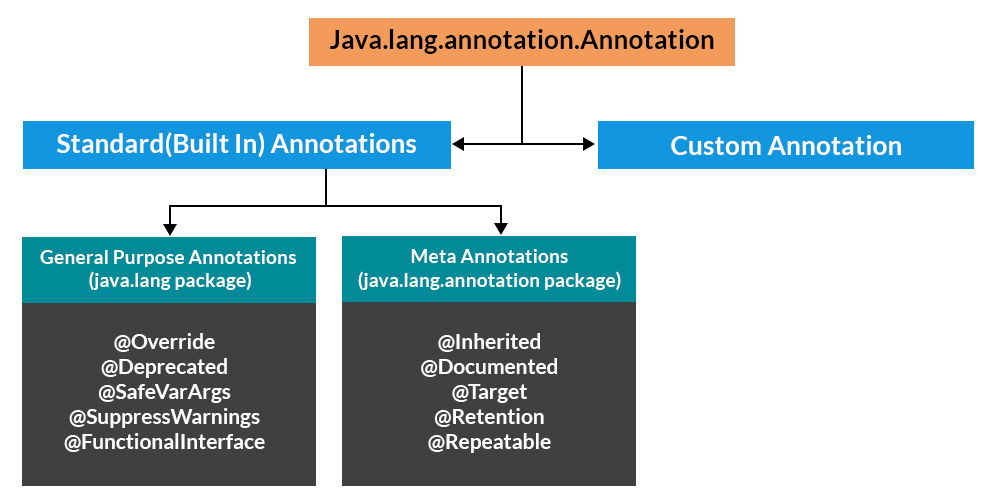
javac --enable-preview --release 20 PatternMatching.java

java --enable-preview PatternMatching

# Annotations

Annotations are used to provide supplemental information about a program.

* Annotations start with ‘**@**’.
* Annotations do not change the action of a compiled program.
* Annotations help to associate *metadata* (information) to the program elements i.e. instance variables, constructors, methods, classes, etc.
* Annotations are not pure comments as they can change the way a program is treated by the compiler. See below code for example.
* Annotations basically are used to provide additional information, so could be an alternative to XML and Java marker interfaces.



There are broadly 5 categories of annotations as listed:

1. Marker Annotations
2. Single value Annotations
3. Full Annotations
4. Type Annotations
5. Repeating Annotations

Let us discuss and we will be appending code wherever required if so.

### **Category 1:**Marker Annotations

The only purpose is to mark a declaration. These annotations contain no members and do not consist of any data. Thus, its presence as an annotation is sufficient. Since the marker interface contains no members, simply determining whether it is present or absent is sufficient. **@Override** is an example of Marker Annotation.

**Example**

@TestAnnotation()

### **Category 2:**Single value Annotations

These annotations contain only one member and allow a shorthand form of specifying the value of the member. We only need to specify the value for that member when the annotation is applied and don’t need to specify the name of the member. However, in order to use this shorthand, the name of the member must be a value.

**Example**

@TestAnnotation(“testing”);

### **Category 3:**Full Annotations

These annotations consist of multiple data members, names, values, pairs.

**Example**

@TestAnnotation(owner=”Rahul”, value=”Class Geeks”)

### **Category 4:**Type Annotations

These annotations can be applied to any place where a type is being used. For example, we can annotate the return type of a method. These are declared annotated with ***@Target****annotation*.

<https://www.geeksforgeeks.org/annotations-in-java/>

<https://youtu.be/ZWJsoX4Cc0E?si=2O5rq_W23hLOaiwO>

# Resources

<https://skillcertpro.com/product/java-se-8-programmer-i-1z0-808-exam-questions/>

<https://youtu.be/u8DpSS8Skv0?si=_54I67zh2XZ0Oujq>