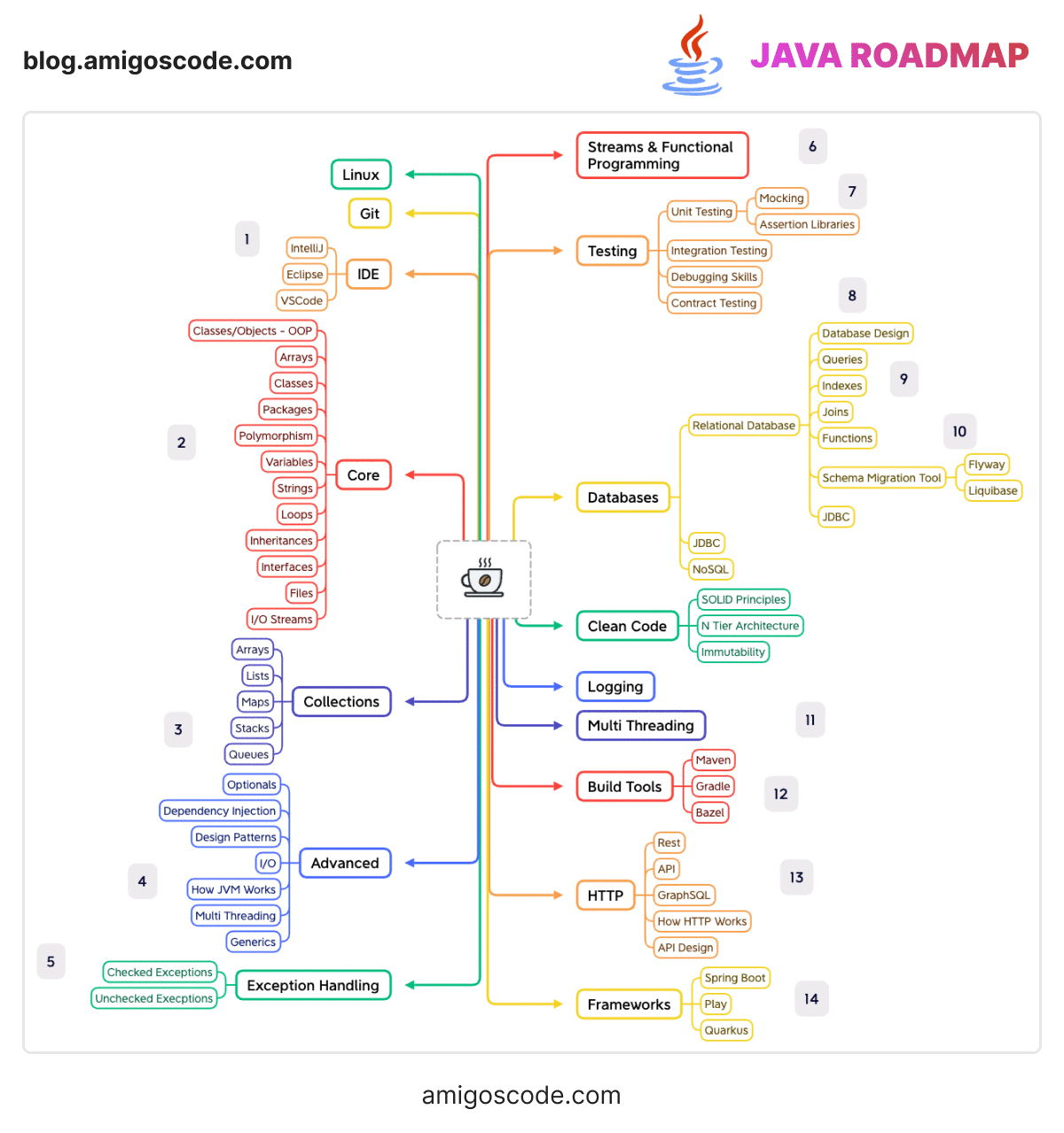
**JAVA LEARNING**



**Java:**

* **Object oriented**
* **Platform independent**: Compiler converts source code to bytecode and then the JVM executes the bytecode generated by the compiler. This bytecode can run on any platform.
* **Portable:** Write Once Run Anywhere
* **Interpreted**
* **Multi-Threaded:** allows concurrent execution of two or more parts of a program for maximum utilization of the CPU.
* **Robust**: The main features of java that make it robust are garbage collection, Exception Handling, and memory allocation.
* **Distributed**: Remote Method Invocation and Enterprise Java Beans are used for creating distributed applications in java.
* **High Performance**: java uses Just In Time (JIT) compiler where the compiler compiles code on-demand basis where it only compiles those methods that are called making applications to execute faster.

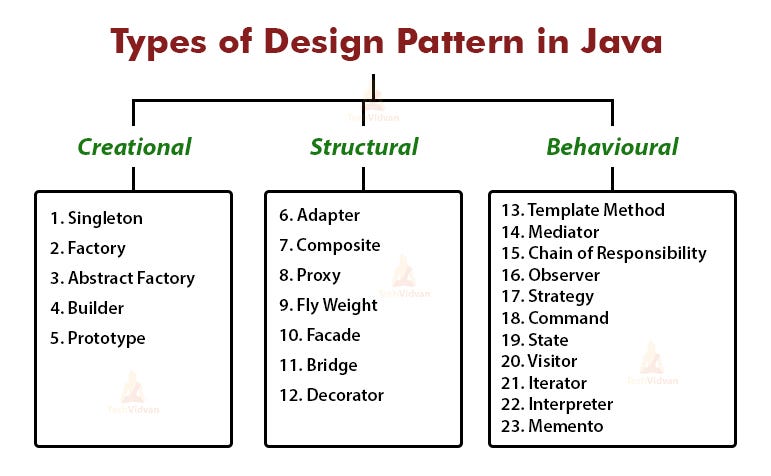
**Disadvantage:**

**Performance:** Java can be slower compared to other programming languages, such as C++, due to its use of a virtual machine and automatic memory management.

The word SOLID acronym for:

* **Single Responsibility Principle (SRP)-** **every Java class must perform a single functionality.**
* **Open-Closed Principle (OCP)-** **the module should be open for extension but closed for modification.**
* **Liskov Substitution Principle (LSP)-** It states that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. In other words, if a class S is a subclass of class T, an object of type T should be replaceable with an object of type S without altering the desirable properties of the program.
* **Interface Segregation Principle (ISP)-** The principle states that the larger interfaces split into smaller ones. Because the implementation classes use only the methods that are required. We should not force the client to use the methods that they do not want to use.
* **Dependency Inversion Principle (DIP)-** The principle states that we must use abstraction (abstract classes and interfaces) instead of concrete implementations. High-level modules should not depend on the low-level module but both should depend on the abstraction.
* Any higher classes should always depend upon the abstraction of the class rather than the detail.
* This aims to reduce the coupling between the classes is achieved by introducing abstraction between the layer, thus doesn’t care about the real implementation.

**Design Patterns:**



**ClassPath:** The classpath is the file path where the java runtime and Java compiler look for .class files to load. By default, JDK provides many libraries. If you want to include external libraries they should be added to the classpath.

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

**class** : class keyword is used to declare classes in Java

**public** : It is an access specifier. Public means this function is visible to all.

**static** : static is again a keyword used to make a function static. To execute a static function you do not have to create an Object of the class. The main() method here is called by JVM, without creating any object for class.

**void** : It is the return type, meaning this function will not return anything.

**main** : main() method is the most important method in a Java program. This is the method which is executed, hence all the logic must be inside the main() method. If a java class is not having a main() method, it causes compilation error.

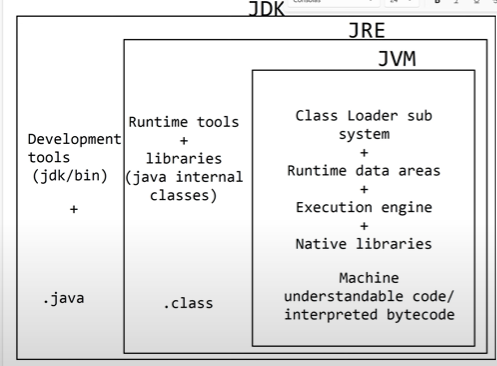
**String[] args** : This is used to signify that the user may opt to enter parameters to the Java Program at command line. We can use both String[] args or String args[]. Java compiler would accept both forms.

**System.in:** This is the standard input stream that is used to read characters from the keyboard or any other standard input device.

**System.out:** This is the standard output stream that is used to produce the result of a program on an output device like the computer screen.

**println():** This method in Java is also used to display text on the console. It prints the text on the console and the cursor moves to the start of the next line at the console. The next printing takes place from the next line.

**JDK in Java:**



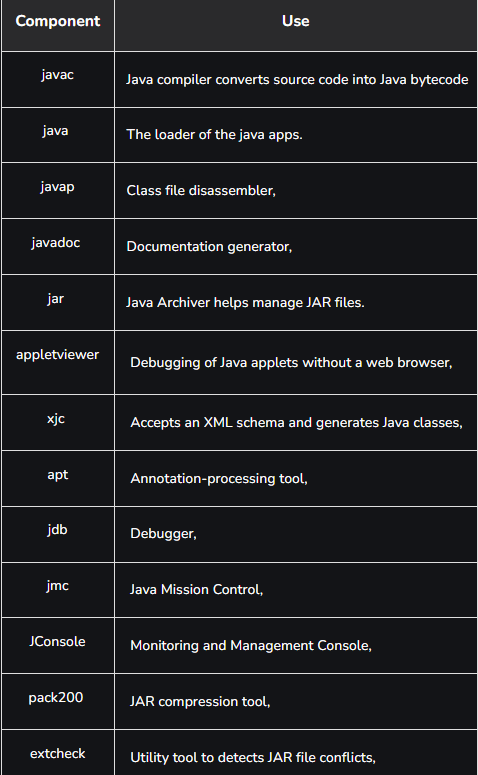
* It is a cross-platformed software development environment that offers a collection of tools and libraries necessary for developing Java-based software applications.

**Contents of JDK:**

* Java Runtime Environment (JRE),
* An interpreter/loader (Java),
* A compiler (javac),
* An archiver (jar) and many more.

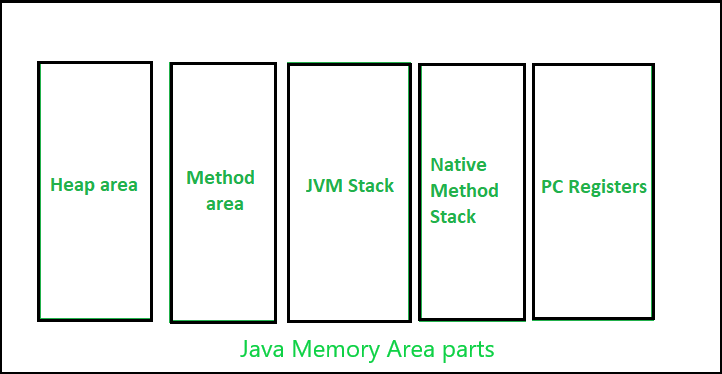
Important Components of JDK

Below there is a comprehensive list of mostly used components of Jdk which are very useful during the development of a java application.





* **All classes and libraries like java.lang, java.util etc. comes from jre/lib/rt.jar.**
* **These libraries are loaded by Bootstrap class loader.**
* **Classes specific for os(Mac, Linux, Windows) are present in lib/est.jar and loaded by Extension class loader.**
* **Application class loader is used to load xml or html files for java execution.**

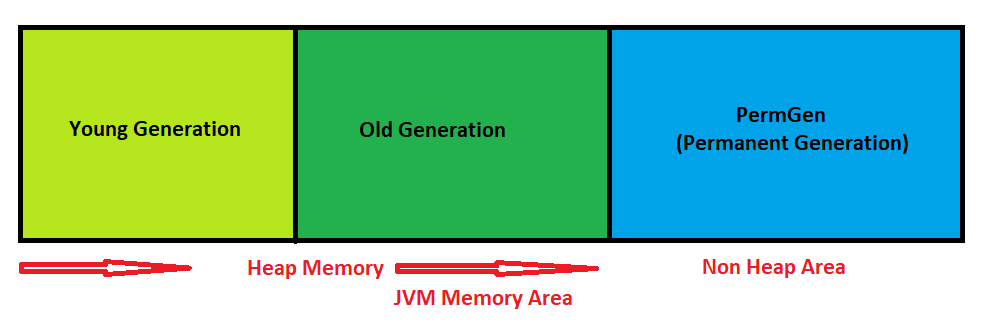


the heap area is one of the most important memory areas of JVM. Here, all the [java objects](https://www.geeksforgeeks.org/classes-objects-java/) are stored. The heap is created when the JVM starts.

The heap is generally divided into two parts. That is: 

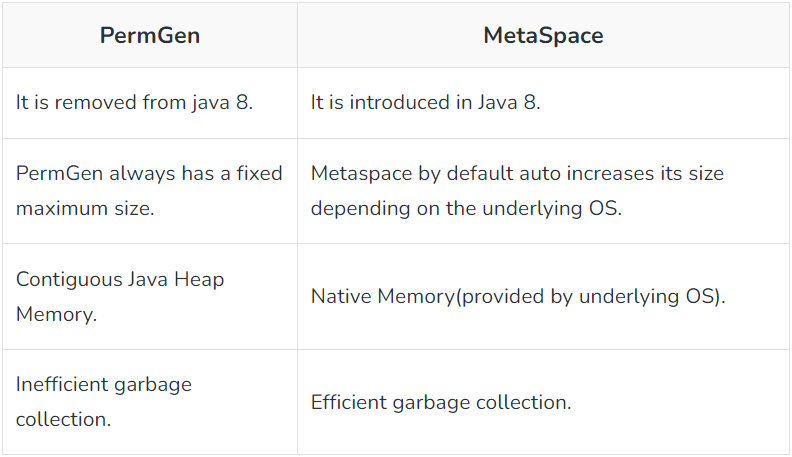
1. **Young Generation(Nursery):** All the new objects are allocated in this memory. Whenever this memory gets filled, the [garbage collection](https://www.geeksforgeeks.org/garbage-collection-java/) is performed. This is called as *Minor Garbage Collection*.
2. **Old Generation:** All the long lived objects which have survived many rounds of minor garbage collection is stored in this area. Whenever this memory gets filled, the garbage collection is performed. This is called as *Major Garbage Collection*.

Apart from the heap memory, JVM also contains another type of memory which is called as “Permanent Generation or PermGen”.



**PermGen Memory:**

* This is a special space in java heap which is separated from the main memory where all the [static content is stored](https://www.geeksforgeeks.org/understanding-storage-of-static-methods-and-static-variables-in-java/).
* Apart from that, this memory also stores the application metadata required by the JVM. Metadata is a data which is used to describe the data.
* Here, garbage collection also happens like any other part of the memory. String pool was also part of this memory before Java 7.
* Method Area is a part of space in the PermGen and it is used to store the class structure and the code for methods and constructors.
* The biggest disadvantage of PermGen is that it contains a limited size which leads to an [OutOfMemoryError](https://www.geeksforgeeks.org/understanding-outofmemoryerror-exception-java/). The default size of PermGen memory is 64 MB on 32-bit JVM and 82 MB on the 64-bit version.
* Due to this, JVM had to change the size of this memory by frequently performing Garbage collection which is a costly operation.
* Java also allows to manually change the size of the PermGen memory. However, the PermGen space cannot be made to auto increase.
* So, it is difficult to tune it. And also, the garbage collector is not efficient enough to clean the memory.
* Due to the above problems, PermGen has been completely removed in Java 8. In the place of PermGen, a new feature called Meta Space has been introduced.
* MetaSpace grows automatically by default.
* Here, the garbage collection is automatically triggered when the class metadata usage reaches its maximum metaspace size.

****

**Garbage Collector:**

* In Java, the Garbage Collector (GC) is a part of the Java Virtual Machine (JVM) responsible for automatic memory management.
* Its primary purpose is to automatically reclaim memory that is no longer in use by the program, preventing memory leaks and ensuring efficient utilization of resources.

Key features and concepts related to the Java Garbage Collector include:

1.Automatic Memory Management:

* Java abstracts the manual memory management present in languages like C and C++.
* Developers do not need to explicitly allocate and deallocate memory.

2.Heap Memory:

* In Java, objects are dynamically allocated on the heap memory.
* The heap is a region of memory used for the storage of objects that are created during the execution of a Java program.

3.Reference Counting vs. Reachability:

* The Java Garbage Collector does not use a reference counting mechanism.
* Instead, it relies on the concept of reachability.
* An object is considered reachable if it can be accessed through references from the root of the object graph (e.g., from the main method, threads, static variables).

4.Types of Garbage Collectors:

* Serial Garbage Collector: A simple, single-threaded collector suitable for small to medium-sized applications.
* Parallel Garbage Collector: Uses multiple threads to improve garbage collection performance on multiprocessor machines.
* Concurrent Mark-Sweep (CMS) Garbage Collector: Designed to minimize application pause times by doing most of the garbage collection work concurrently with the application threads.
* G1 Garbage Collector: Introduces the Garbage-First approach, aiming to provide high throughput and low-latency garbage collection.

5.Major Phases of Garbage Collection:

* Mark: Identify and mark reachable objects.
* Sweep: Reclaim memory occupied by unreachable objects.
* Compact (optional): Compacts memory to reduce fragmentation (not always performed in all collectors).

6.Tuning and Configuration:

* Developers can configure and tune the garbage collector based on application requirements, using command-line options or configuration files.

7.Memory Management Best Practices:

* Properly managing object lifetimes and avoiding unnecessary object creation contribute to efficient garbage collection.

8.Finalization:

* The finalize method in Java allows objects to perform cleanup operations before being garbage collected.
* However, relying on finalize is discouraged, and modern applications prefer using other resource management mechanisms.

**The Java Garbage Collector helps Java developers by handling memory management automatically, reducing the risk of memory-related issues, and providing a more developer-friendly environment.**

**OOPS:**

* A class in Java is a template or blueprint that defines the structure and behavior of objects.
* It specifies the attributes (data) and methods that objects of that class will have. An object, on the other hand, is an instance of a class.
* It represents a specific entity or instance created based on the class blueprint.

An object consists of :

**State**: It is represented by the attributes of an object. It also reflects the properties of an object.

**Behaviour**: It is represented by the methods of an object. It also reflects the response of an object to other objects.

**Identity**: It gives a unique name to an object and enables one object to interact with other objects.

**1.Encapsulation:**

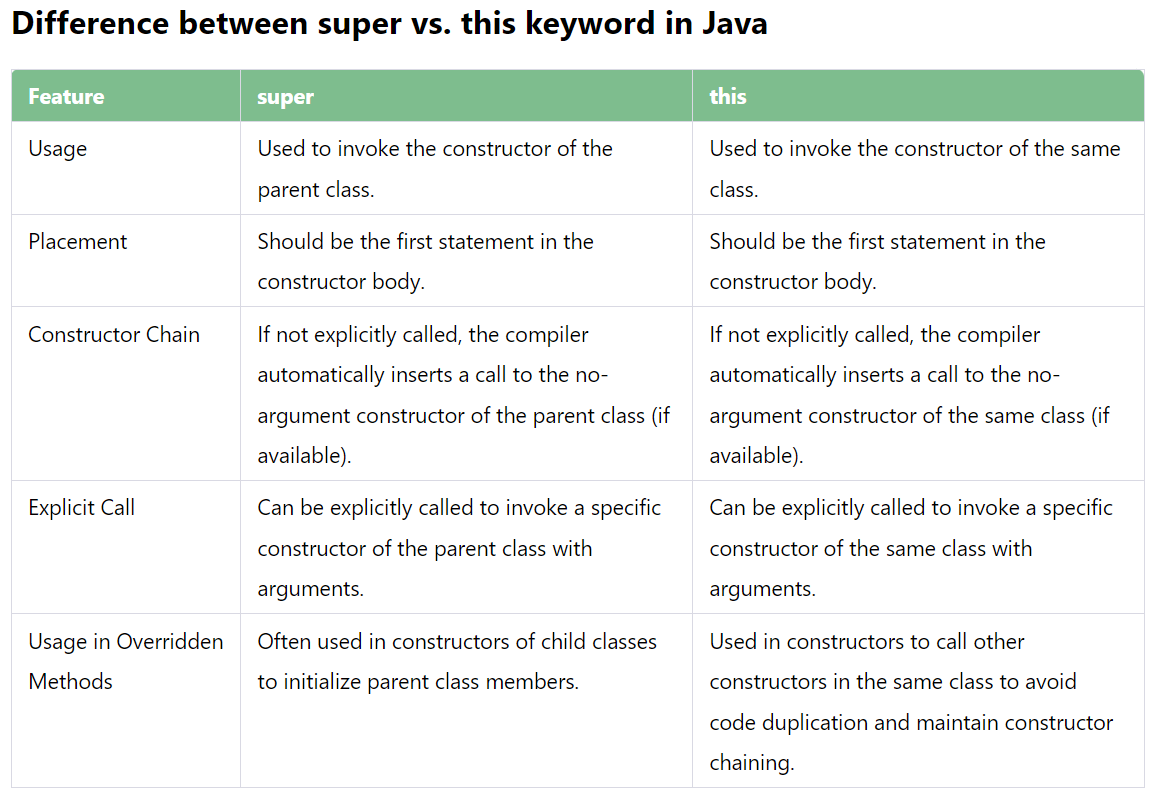
* It is the process of wrapping code and data together into a single unit within a class and restricts access to the data by using access modifiers.
* It allows for data hiding, protecting the internal state of an object and ensuring that it can only be accessed through defined setter and getter methods.
* In Java, encapsulation is achieved using access modifiers (private, protected, public).
* In order to achieve encapsulation in java follow certain steps as proposed below:

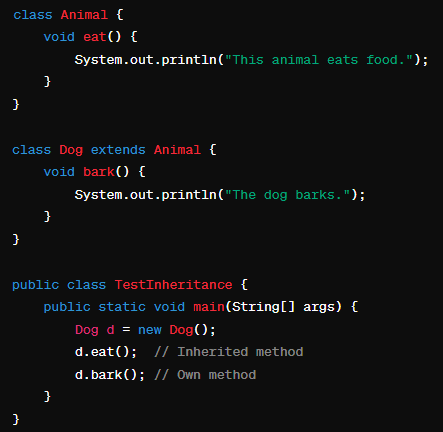
1.Declare the variables as private

2.Declare the setters and getters to set and get the variable values

**2.Inheritance:**

* It is a mechanism in Java that allows a class to inherit properties and behaviors from another class.
* It promotes code reuse by enabling the creation of subclasses that inherit the attributes and methods of a superclass.
* Subclasses can also add their own unique attributes and methods.
* Inheritance is used when we have is-a relationship between objects. Inheritance in Java is implemented using extends keyword.



****

**3.Abstraction:**

* Abstraction is a process of hiding implementation details and exposes only the functionality to the user.
* In abstraction, we deal with ideas and not events. This means the user will only know “what it does” rather than “how it does”.
* There are two ways to achieve abstraction in Java:

1. Abstract class (0 to 100%)
2. Interface (100%)

**Abstract Classes:**

* The class should be abstract if a class has one or many abstract methods.
* An abstract class can have constructors, concrete methods, static method, and final method.
* Abstract class can’t be instantiated directly with the new operator.

**Ex: A b = new B();**

* Abstract class in Java is valid even if it does not contain any abstract methods.
* The primary purpose of an abstract class is to serve as a superclass that cannot be instantiated on its own.
* This allows you to define a common base for related classes that share some implementation but also require that subclasses provide specific implementations.
* Abstract classes in Java cannot be instantiated directly and are typically used as blueprints for creating objects.
* They can contain abstract methods (methods without implementation) and regular methods.
* Abstract classes provide a way to define common behavior and enforce specific methods to be implemented by subclasses.

**Interface:**

* **Abstract Methods Only (Before Java 8)**

- All methods in an interface are implicitly public and abstract.

* **Default and Static Methods (Since Java 8)**

- Default methods provide a default implementation that can be overridden by implementing classes.

- Static methods are similar to static methods in classes; they belong to the interface and can be called independently of any object.

* **Constant Variables**

**-** It is implicitly public, static, and final.

* **Multiple Inheritance**
* **Interface Inheritance**

**-** Interfaces can extend other interfaces, allowing the creation of a hierarchy of interfaces.

* **Functional Interfaces (Since Java 8)**

**-** A functional interface is an interface with a single abstract method (SAM). Functional interfaces can be used as lambda expressions or method references.

**4.Polymorphism:**

* It refers to the ability of an object to take on many forms or have multiple behaviors.
* In Java, polymorphism is achieved through **method overriding and method overloading.**

**Method Overriding:**

* Method overriding is a feature in Java that allows a subclass to provide a specific implementation for a method that is already defined in its superclass.
* When a method in a subclass has the same name, return type, and parameters as a method in its superclass, the subclass's method overrides the superclass's method.
* It allows objects of different classes to be treated as objects of a common superclass, providing flexibility and extensibility.
* The method in the subclass must have the same name, return type, and parameters as the superclass method.
* **Upcasting**: An instance of a subclass (e.g., Dog or Cat) is referred to by a reference variable of the superclass type (Animal). This is known as upcasting.
* **Run-time Resolution:** The method to be executed is determined at run time based on the actual object type that the superclass reference variable points to. This enables dynamic method dispatch.

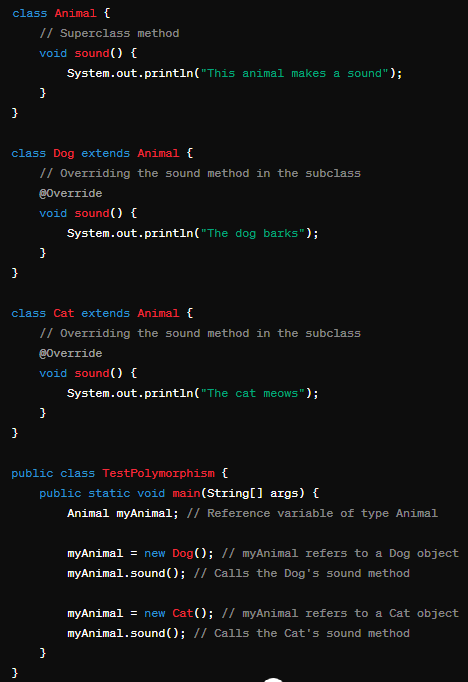
**Rules for Method Overriding:**

**1.Same Method Signature:** The method in the subclass must have the same name, return type, and parameters as the method in the superclass.

**2.Access Level:** The overridden method must not be more restrictive. For example, if the superclass method is public, the subclass method cannot be protected or private.

**3.Exceptions:** The subclass method should not throw new or broader checked exceptions than the overridden method.

**4.Instance Methods:** Only instance methods can be overridden. Static methods cannot be overridden, but they can be hidden (this is called method hiding, not overriding).

****

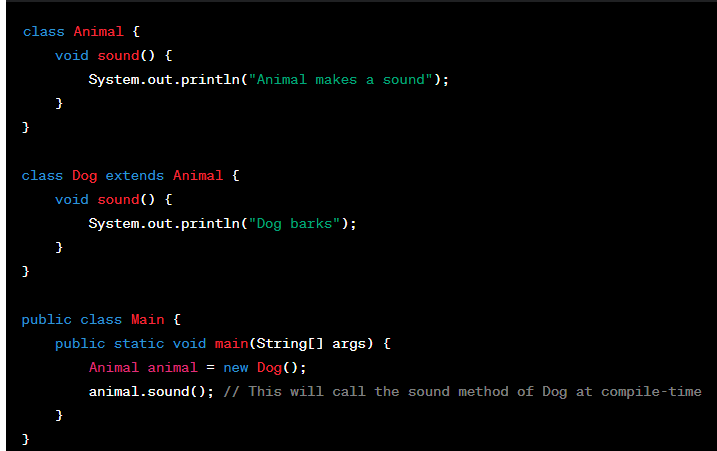
**Binding:**

* In Java, there are two main types of binding: static binding (also known as early binding) and dynamic binding (also known as late binding).
* These concepts are associated with how the association between a method call and the method implementation is resolved in a program.

**1. Static Binding (Early Binding):**

* Static binding occurs during the compile-time.
* The compiler determines the method or field to be called based on the type of the reference variable at compile-time.
* This binding is also known as compile-time binding because it is resolved during the compilation phase.
* Faster execution because the binding is done at compile time.
* Even though the reference variable animal is of type Animal, the method call animal.sound() is resolved at compile-time to the sound method of the Dog class because the actual object is of type Dog.

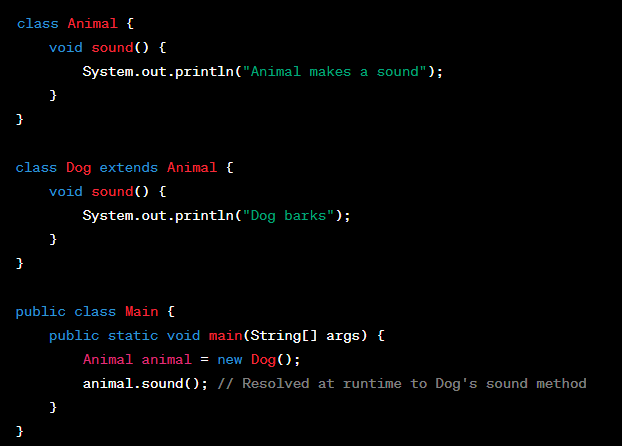
**Ex:**

****

**2. Dynamic Binding (Late Binding):**

* Dynamic binding occurs during the runtime.
* The method or field to be called is determined at runtime based on the actual type of the object.
* The JVM determines the type of the actual object at runtime and then resolves the method call.
* This is also known as runtime polymorphism.

Ex:



**we can create objects in Java.**

1. **Using new keyword:**

// Define a simple class

class MyClass {

int value;

// Constructor

public MyClass(int value) {

this.value = value;

}

}

// Create an object using the new keyword

MyClass myObject = new MyClass(42);

1. **Using new instance:**

// Define a class

class Person {

String name;

// Constructor

public Person(String name) {

this.name = name;

}

}

**try {**

**// Create an object using new instance**

**Person personObject = Person.class.newInstance();**

**personObject.name = "John";**

**} catch (InstantiationException | IllegalAccessException e) {**

**e.printStackTrace();**

**}**

1. **Using clone() method:**

// Define a class that implements Cloneable

class Point implements Cloneable {

int x, y;

// Constructor

public Point(int x, int y) {

this.x = x;

this.y = y;

}

// Clone method

public Point clone() {

try {

return (Point) super.clone();

} catch (CloneNotSupportedException e) {

e.printStackTrace();

return null;

}

}

}

// Create an object using object cloning

Point originalPoint = new Point(5, 10);

Point clonedPoint = originalPoint.clone();

1. **Using deserialization:**

import java.io.\*;

// Define a class Serializable

class MyClass implements Serializable {

int value;

// Constructor

public MyClass(int value) {

this.value = value;

}

}

// Serialize the object

MyClass originalObject = new MyClass(42);

try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("object.ser"))) {

out.writeObject(originalObject);

} catch (IOException e) {

e.printStackTrace();

}

// Deserialize the object

try (ObjectInputStream in = new ObjectInputStream(new FileInputStream("object.ser"))) {

MyClass deserializedObject = (MyClass) in.readObject();

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

1. **Using newInstance() method of Constructor class:**

import java.lang.reflect.Constructor;

// Define a class

class MyClass {

int value;

// Constructor

public MyClass(int value) {

this.value = value;

}

}

try {

// Get the constructor using reflection

Constructor<MyClass> constructor = MyClass.class.getConstructor(int.class);

// Create an object using newInstance method

MyClass myObject = constructor.newInstance(42);

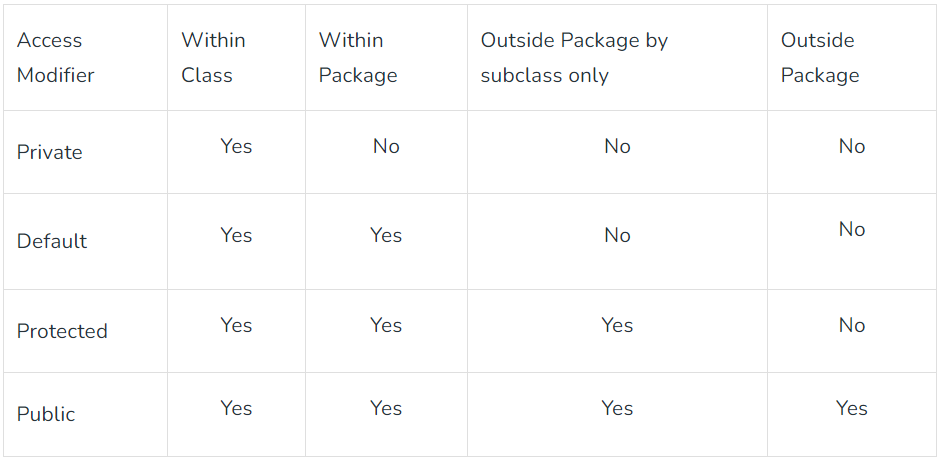
} catch (Exception e) {

e.printStackTrace();

}

**Access Modifiers:** These modifiers control the scope of class and methods.

* Access Modifiers: default, public, protected, private.
* Non-access Modifiers: final, abstract, static, transient, synchronized, volatile, native.



Public > protected > default > private

Implementation of a Java application program involves a following step.

They include:

1. Creating the program

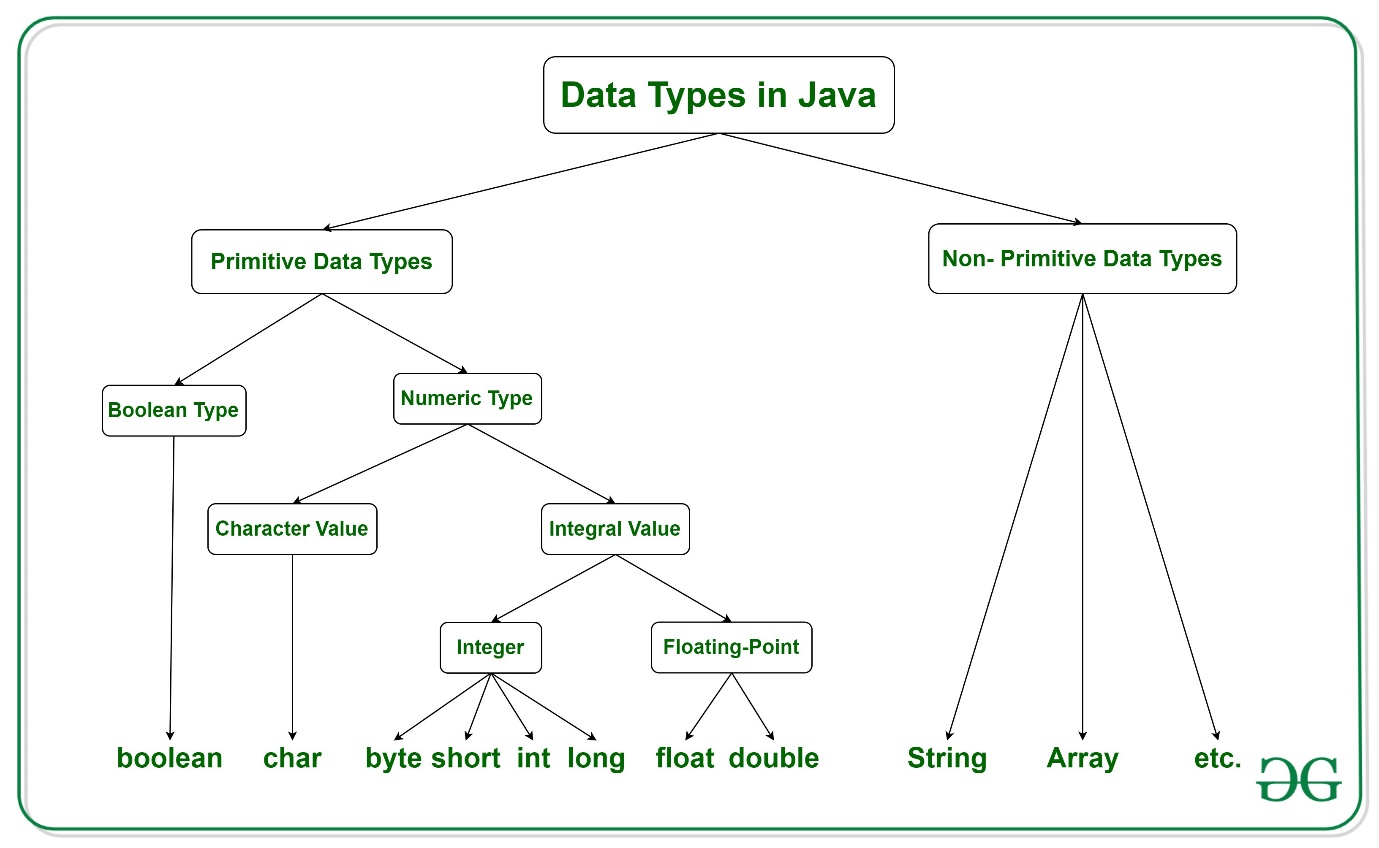
2. Compiling the program

3. Running the program

**Data Types:**

1. Primitive Data Type: such as boolean, char, int, short, byte, long, float, and double
2. Non-Primitive Data Type or Object Data type: such as String, Array, etc.

* The direct superclass of an array type is Object.
* Every array type implements the interfaces Cloneable and java.io.Serializable.
* Char uses 2 bytes in java because it uses the Unicode system rather than the ASCII system.



**Enum:**

* It is introduced in Java 5, is a special data type that consists of a set of pre-defined named values separated by commas.
* These named values are also known as elements or enumerators or enum instances.
* Since the values in the enum type are constant, you should always represent them in UPPERCASE letters.
* You can use an Enum type when you need a fixed set of pre-defined constant values that are known at the compile-time itself. Examples can be days of the week, seasons of the year, etc.
* Like Java Classes, Enums can contain Constructors, Member Variables, Methods and also implement Interfaces.
* The set of pre-defined constant values must always be in the very first line. And then, these are followed by constructors, member variables, and methods.
* However, the only difference is that Enum constants are public, static, and final.
* Besides, you can instantiate it using the ‘new’ keyword or extend any other class explicitly.

The following characteristics make enum a ‘special’ class:

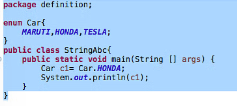
* enum constants cannot be overridden
* enum doesn’t support the creation of objects
* enum can’t extend other classes
* enum can implement interfaces like classes

Syntax:

In Java, to define an Enum type, the ‘enum’ keyword is used rather than ‘class’ or ‘interface’. Mentioned below is the syntax.

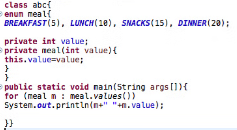
**enum Variable\_Name { VALUE\_1, VALUE\_2, VALUE\_3, … }**

Ex:



**How To Initialize Specific Values To Java Enums With Example ?**

* Enum in Java can have constructors, methods, and fields.
* The initial value of enum constants starts from 0.
* But by defining constructors and fields, it is possible to initialize the specific value to the enum constants.
* The following example explains this:



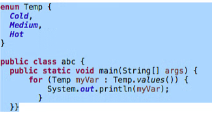
Output:



**Looping Through Enum:**

The values() method of enum in Java is useful for looping through an enum's constants.

**Ex:**



Output:



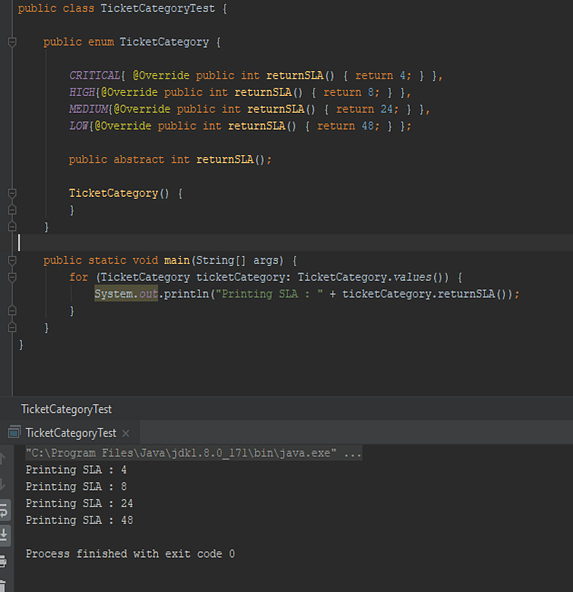
**Standard Enum Methods:**

1. values(): This method is used to get an array of all predefined constant values in an Enum type.
2. ordinal(): Each constant value in enum contains an internal reference number that corresponds to the order in which they are declared, starting with 0. By using this method, you can get the internal reference number for each constant value in Enum.
3. valueOf(): This method is used to return an enum constant of the specified string value if it exists.
4. toString(): This method is used to get a String value for the given enum constant.
5. compareTo()- This method compares the constants of enum on the basis of their ordinal values. For example: meal.LUNCH.compareTo(meal.DINNER) returns ordinal(LUNCH)- ordinal(DINNER)
6. name()-This method returns the defined term of an enum constant in the form of a string. The returned value is FINAL. Example: name(LUNCH) returns “LUNCH”

**Abstract Methods in Enum:**

* You can also have abstract methods inside Enum type.
* If there’s an abstract method, then each enum constant value should implement it.
* It is very useful when you want a different implementation of the method for each constant value.

Ex:



**Operators:**

There are multiple types of operators in Java all are mentioned below:

Arithmetic Operators (+,-,\*,/,%)

Unary Operators(-,+,++,--,!)

Assignment Operator(=,+=,-=,\*=,/=,%=)

Relational Operators(==,<,>,>=,<=)

Logical Operators (&&,||,!)

Ternary Operator(used for if-else : condition ? if true : if false)

Bitwise Operators (& - and,| - or,^ - xor,~ - complement)

Shift Operators (<< , >>)

Left shift <<: 10<<1 = 20

x\*2^n

10\*2^1 = 20

Right shift >> : 10>>1 = 5

instance of operator (To know if obj belongs to a class, subclass or interface.

object name **instance of** class/sub class/ interface name)

**Variables in Java:**

Java variable is a name given to a memory location. It is the basic unit of storage in a program.

The value stored in a variable can be changed during program execution.

Variables in Java are only a name given to a memory location. All the operations done on the variable affect that memory location.

In Java, all variables must be declared before use.

Best Practices:

* No single letter.
* Always be specific
* Ideally 1-2 words
* Use ‘is’ prefix for Boolean
* Use camelCase
* Use ALL\_CAPS with underscores for constants.

1. Local Variables:

* **A variable defined within a block or method or constructor is called a local variable.**
* These variables are created when the block is entered, or the function is called and destroyed after exiting from the block or when the call returns from the function.
* Scope – only where it is declared.
* Initialize variable mandatory.

{

Ex: **int** x = 10;

}

2. Instance Variables:

* **Instance variables are non-static variables and are declared in a class outside of any method, constructor, or block.**
* These variables are created when an object of the class is created and destroyed when the object is destroyed.
* We may use access specifiers.
* Initialization of an instance variable is not mandatory (default value dependent on data type of variable).
* **Accessible only through objects. Each object will have its own copy of an instance variable.**
* Can be initialized using constructor or instance blocks.

Ex:  **public** String geek;

3. Static Variables

* Static variables are also known as class variables.
* **The static variables are declared using the static keyword within a class outside of any method, constructor, or block.**
* **Only one copy of static variables per class.**
* Static variables are created at the start of program execution and destroyed automatically when execution ends.
* Initialization of an instance variable is not mandatory (default value dependent on data type of variable).
* Static blocks can be used to initialize static variables.
* If we access a static variable like an instance variable (through an object), the compiler will show a warning message, which won’t halt the program. The compiler will replace the object name with the class name automatically.
* If we access a static variable without the class name, the compiler will automatically append the class name.

Ex:  **public** **static** String geek = "Shubham Jain";

We can access instance variables through object references, and static variables can be accessed directly using the class name.

Instance variables are created when an object is created with the use of the keyword ‘new’ and destroyed when the object is destroyed. Static variables are created when the program starts and destroyed when the program stops.

In Java, when a class is loaded and initialized, the order of execution follows a specific sequence involving static blocks, instance blocks, constructors, and the main method print statements. Let's break down this sequence:

**1.Static Block:**

The static block is executed first because it's executed when the class is loaded into memory. In your case, the static block prints "Static block".

**2.Main Method Print Statement:**

The main method is the entry point of the Java program. When you run the program, the main method is invoked, printing "Main method print statement".

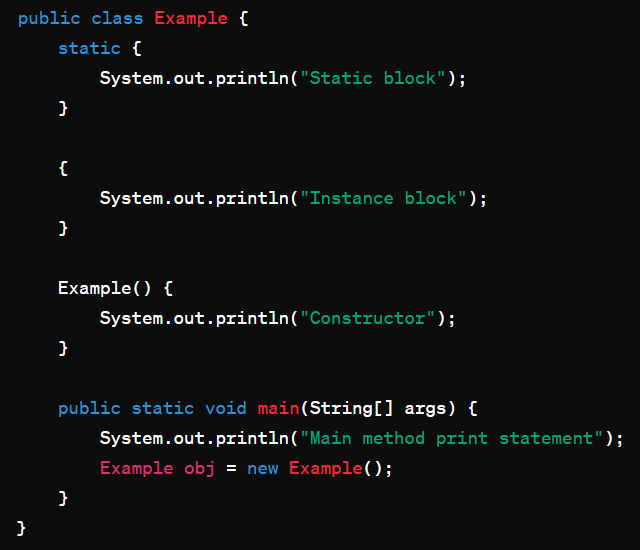
**3.Instance Block:**

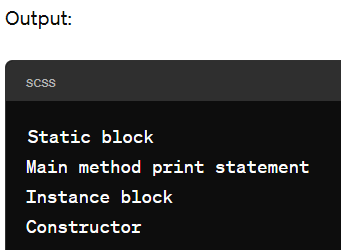
After the main method print statement, an instance of the Example class is created (new Example()). When an instance is created, instance blocks are executed before the constructor. Therefore, the instance block prints "Instance block".

**4.Constructor:**

Finally, the constructor of the Example class is executed after the instance block, printing "Constructor".

Here's a simplified example to illustrate the order of execution:





**Wrapper Classes:**

* A Wrapper class in Java is a class whose object wraps or contains primitive data types.

Need of Wrapper Classes:

* They convert primitive data types into objects.
* The classes in java.util package handles only objects and hence wrapper classes help in this case also.
* Collection framework stores only objects not primitive types.
* Only objects support cloning, serialization, and synchronization in multithreading.
* **Autoboxing** – conversion of primitive types to object (int to Integer).

Ex: char ch = ‘a’;

Character c= ch;

Or

int b = 10;

Integer intobj = new Integer(b);

* **Unboxing** – conversion of object to resp. primitive type(Integer to int).

Ex: Character ch = ‘a’;

Char c = ch;

Or

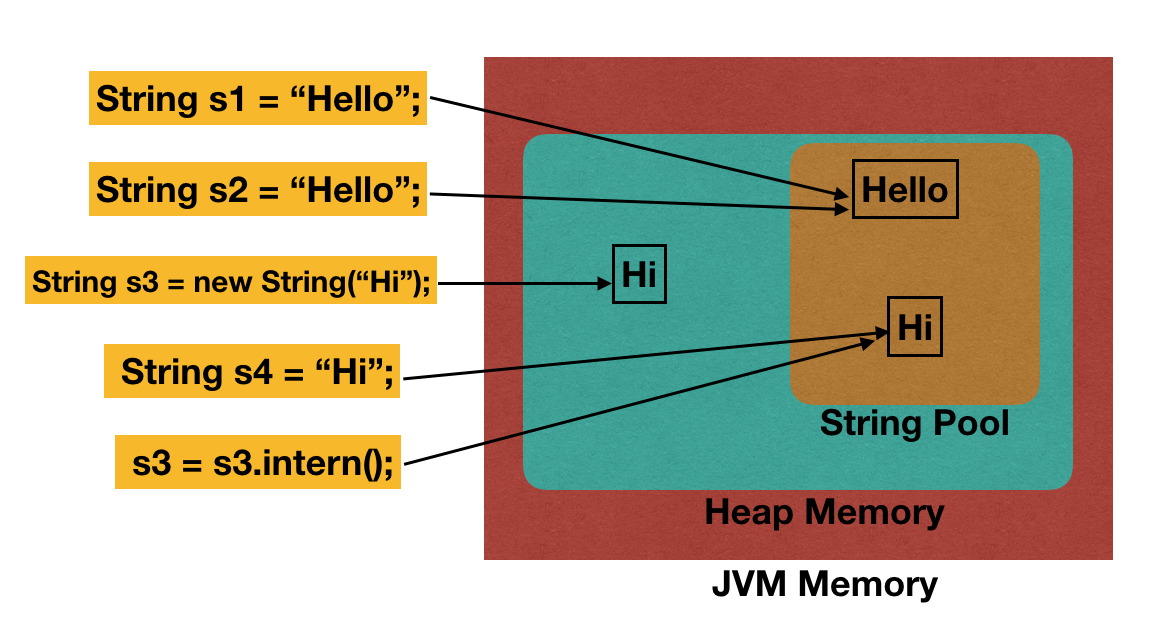
**int** iv = intobj;

**String:**

* If a string is going to remain constant throughout the program, then use the String class object because a String object is immutable.
* If a string can change and will only be accessed from a single thread, using a StringBuilder is good enough.
* If a string can change and will be accessed from multiple threads, use a StringBuffer because StringBuffer is synchronous, so you have thread-safety.
* If you don’t want thread-safety than you can also go with StringBuilder class as it is not synchronized.

**How Does Java String Pool Works?**

* When we create a string literal, it’s stored in the string pool.
* If there is already a string with the same value in the string pool, then new string object is not created. The reference to the existing string object is returned.
* Java String Pool is a cache of string objects. It’s possible because string is immutable.
* If we create a string object using new operator, it’s created in the heap area. If we want to move it to the string pool, we can use intern() method.



**Serialization & deserialization:**

<https://chatgpt.com/share/6734ec9a-3e58-8010-af58-630c9432882c>

* Serialization is the process of converting an object’s state into a byte stream.
* This byte stream can then be saved to a file, transferred over a network, or stored in a database.
* The main goal is to "persist" an object’s data so it can be reused or transferred.

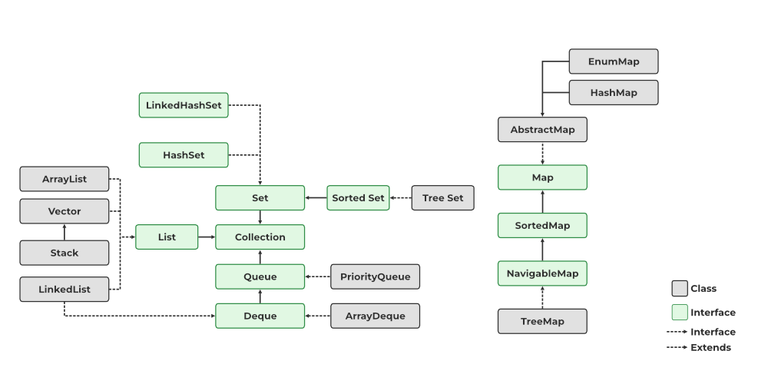
**Uses:**

* Persistence: Saving the state of an object to a file or database for later retrieval.
* Caching: Serializing objects to cache complex computations or frequently accessed data for faster retrieval.
* Communication in Distributed Systems: Transmitting objects over a network between servers in a client-server architecture or within microservices.

**Cons:**

* Inefficiency and Performance: Serialization can be slow and consume more memory compared to other data formats (like JSON or XML).
* Security Risks: Serialized objects can expose sensitive data and are vulnerable to attacks if not handled carefully, as arbitrary data can be injected into deserialization streams.
* Lack of Compatibility: Serialization creates JVM-dependent data, making it challenging to deserialize Java objects in other languages without additional processing.
* Complexity with Changes: Modifying a serialized class (adding/removing fields) can lead to compatibility issues, requiring careful management with serialVersionUID.

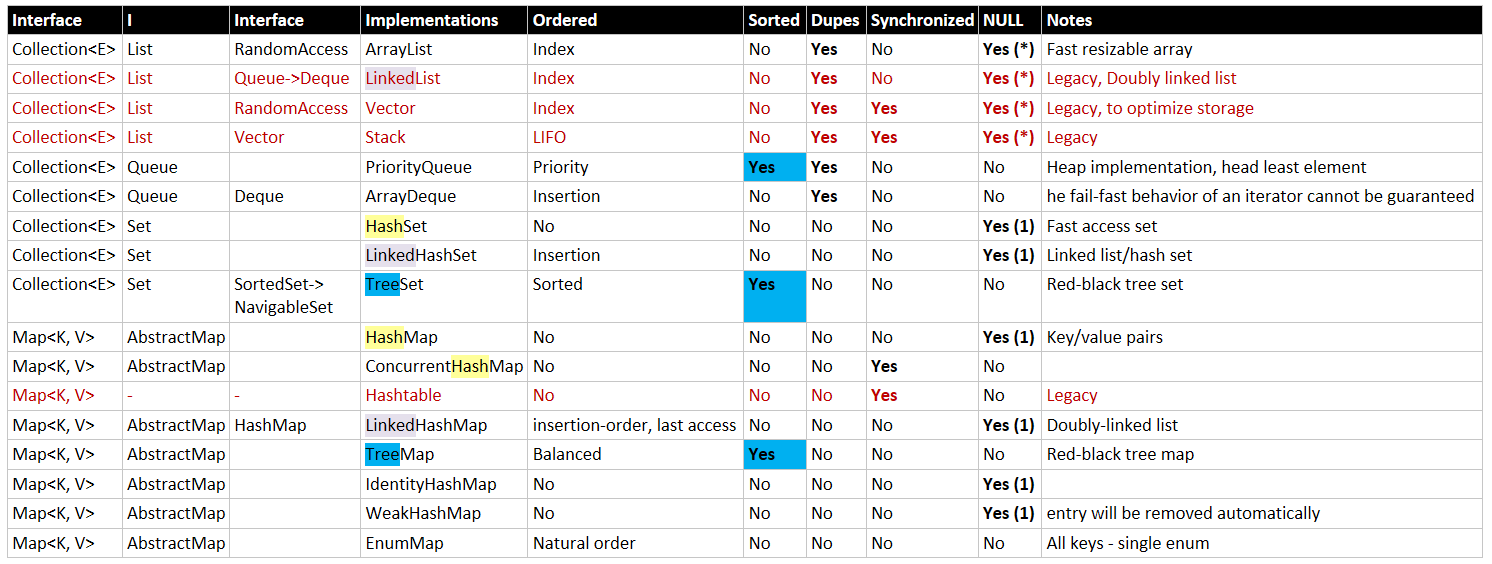
**Collections:**



**Consistent API:**

* The API has a basic set of interfaces like Collection, Set, List, or Map, all the classes (ArrayList, LinkedList, Vector, etc) that implement these interfaces have some common set of methods.
* The utility package, (java.util) contains all the classes and interfaces that are required by the collection framework.
* The collection framework contains an interface named an iterable interface which provides the iterator to iterate through all the collections.
* Iterable Interface: 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.



**List(ArrayList,LinkedList,Vector,Stack):**

**Operations:**

* Create
* Add elements
* Retrieve elements
* Delete elements
* Search elements(present or not)
* Update elements

**ArrayList:(asynchronous, ordered & allows null & duplicates):**

* ArrayList provides us with dynamic arrays in Java.
* slower than standard array but helps to do lots of manipulation in the array.
* ArrayList can not be used for primitive types, like int, char, etc.
* We will need a wrapper class for such cases. better for storing and accessing data.
* Fast random access.
* Slower insertion and deletion in the middle of the list.
* Less memory overhead.
* Suitable for read-heavy operations.

**LinkedList:(ordered, allows duplicates ):**

* A linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part.
* Default size will be zero.
* The elements are linked using pointers and addresses.
* Each element is known as a node.
* It can be used as list, stack or queue.
* better for manipulating data.
* Slower random access.
* Faster insertion and deletion at the ends or using an iterator.
* More memory overhead.
* Suitable for write-heavy operations where elements are frequently added or removed.

**vector:(synchronous):**

* A vector provides us with dynamic arrays in Java.
* It is a synchronized ArrayList.
* Java Vector contains many legacy methods that are not the part of a collections framework.
* vector default size is 10.

**Stack:(synchronous):**

* The class is based on the basic principle of last-in-first-out.
* stack extends Vector class.
* In addition to the basic push and pop operations, the class provides three more functions empty, search, and peek.

**Queue:(ordered, synchronous):**

**(** **PriorityQueue, ArrayDequeue)**

* PriorityQueue and LinkedList are not thread-safe.
* The Queue interface is inherited by 3 main sub-interfaces. Blocking Queues, Transfer Queues, and Deques.
* A queue interface maintains the FIFO(First In First Out) order similar to a real-world queue line.
* This interface is dedicated to storing all the elements where the order of the elements matter.
* The most frequently used implementation of the queue interface is the PriorityQueue.
* A PriorityQueue is used when the objects are supposed to be processed based on priority.
* It is known that a queue follows the First-In-First-Out algorithm, but sometimes the elements of the queue are needed to be processed according to the priority and this class is used in these cases.
* Deque, also known as a double-ended queue, is a data structure where we can add and remove elements from both ends of the queue.
* The class which implements the deque interface is ArrayDeque.

**set:(no duplicates):**

**(HashSet, TreeSet(sorted), LinkedHashSet, EnumSet)**

* A set is an unordered collection of objects in which duplicate values cannot be stored.

Set<T> hs = new HashSet<> ();

Set<T> lhs = new LinkedHashSet<> ();

Set<T> ts = new TreeSet<> ();

* hashset: The objects that we insert into the HashSet do not guarantee to be inserted in the same order.
* The objects are inserted based on their hashcode.
* This class also allows the insertion of one NULL elements.
* A LinkedHashSet is very similar to a HashSet.
* The difference is that this uses a doubly linked list to store the data and retains the ordering of the elements.
* sorted set interface extends the set interface and is used to handle the data which needs to be sorted.
* The class which implements the sorted set interface is TreeSet.

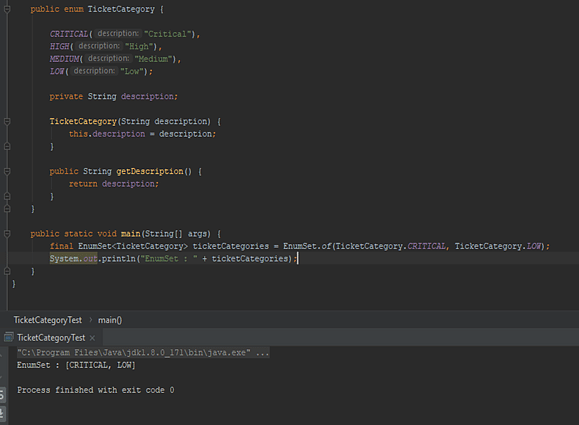
SortedSet<T> ts = new TreeSet<> ();

* The TreeSet class uses a Tree for storage.
* The ordering of the elements is maintained by a set using their natural ordering whether or not an explicit comparator is provided.

**EnumSet:**

* This abstract class is a specialized implementation of the Java Set that can be used with Enum types.
* This is a more efficient, high-quality, type-safe alternative to HashSet implementation.
* All the elements in this EnumSet must contain the same enum type only.
* This is not synchronized and doesn’t allow Null elements.
* If you try to insert a Null element, it will throw a NullPointerException.

Ex:



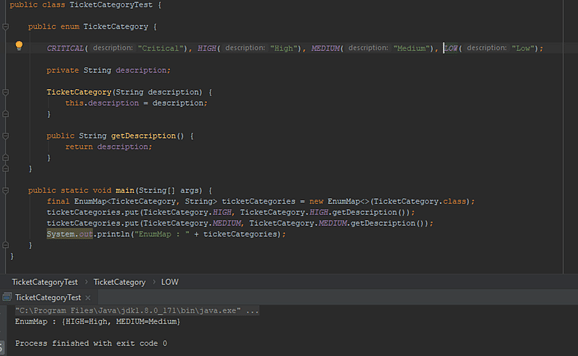
**map:**

**(HashMap,TreeMap(async), EnumMap)**

Map<T> hm = new HashMap<> ();

Map<T> tm = new TreeMap<> ();

* The frequently used implementation of a Map interface is a HashMap.
* A map is a data structure that supports the key-value pair for mapping the data.
* This interface doesn’t support duplicate keys because the same key cannot have multiple mappings, however, it allows duplicate values in different keys.
* **HashMap** provides the basic implementation of the Map interface of Java.
* It stores the data in (Key, Value) pairs.
* To access a value in a HashMap, we must know its key.
* HashMap uses a technique called Hashing.
* Internally, a HashMap uses an array-based data structure and a hash table algorithm to store and manage its elements.
* Hashing is the process of translating a given key into a code.
* A hash function is used to substitute the information with a newly generated hash code.
* The underlying data structure of a HashMap is an array of buckets.
* Each bucket is essentially a linked list that stores key-value pairs with the same hash code.
* If multiple keys have the same hash code, they are stored in the same bucket as a linked list.
* HashSet also uses Hashtable internally.
* **TreeMap** is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.
* This proves to be an efficient way of sorting and storing the key-value pairs and doesn't allow null keys but allows null values.
* A TreeMap is implemented using a Red-Black tree, which is a type of self-balancing binary search tree.
* **EnumMap :** All the keys of the EnumMap must be of the same single enum type.
* ‘Null’ keys are not allowed.
* ‘Null’ values are allowed.
* If you try to insert a Null key, it will throw a NullPointerException.



**Concurrent Collections:**

**1. ConcurrentHashMap:**

* Multiple readers can access the Map concurrently while a portion of Map gets locked for write operation depends upon concurrency level of Map.
* Aternative to synchronized Map or Hashtable.

**Internal Working:**

* ConcurrentHashMap is a thread-safe implementation of the Map interface in Java, designed to support concurrent access by multiple threads without the need for external synchronization. Here's how it works internally:
* **Segmentation:** ConcurrentHashMap is internally divided into multiple segments, each of which acts as a separate hash table. The number of segments is determined by the concurrency level specified during construction or by default if not specified.
* **Hashing:** Like other hash table-based data structures, ConcurrentHashMap uses a hash function to map keys to corresponding segments. Each segment has its own hash table, and the hash code of the key determines which segment the key-value pair belongs to.
* **Segment Locking:** Each segment in ConcurrentHashMap is independently locked using a separate lock, typically implemented as a Reentrant Lock or similar synchronization primitive. This means that multiple threads can access different segments concurrently without blocking each other.
* **Segment-wise Operations**: When performing read or write operations on ConcurrentHashMap, the appropriate segment is determined based on the hash code of the key. For example, when adding a key-value pair, the hash code of the key is used to find the segment, and then the corresponding lock for that segment is acquired to ensure thread safety during the update.
* **Read Operations:** Read operations (e.g., get()) can be performed concurrently without locking because each segment is independently accessible by multiple threads. However, read operations may need to wait if a segment is being modified by a write operation.
* **Write Operations:** Write operations (e.g., put(), remove()) require acquiring the lock for the corresponding segment to ensure mutual exclusion. However, ConcurrentHashMap employs an optimistic approach known as lock striping to minimize contention. Only the affected segment is locked during a write operation, allowing other threads to concurrently access different segments.
* **Expansion and Resizing:** ConcurrentHashMap dynamically resizes itself to accommodate more elements when needed. Expansion is performed segment-wise to minimize the impact on concurrent operations. During resizing, ConcurrentHashMap may also rehash elements to distribute them more evenly across segments.
* **Concurrent Modifications:** ConcurrentHashMap supports weakly consistent iterators, meaning that they reflect the state of the ConcurrentHashMap at some point in time and may not reflect recent updates. Concurrent modifications (e.g., adding or removing elements) can be safely performed while iterating over ConcurrentHashMap without causing ConcurrentModificationException.
* Overall, ConcurrentHashMap achieves thread safety and concurrency by dividing the underlying data structure into multiple segments and using fine-grained locking at the segment level to allow concurrent read and write access. This design enables high performance and scalability in multi-threaded environments.

**2. CopyOnWriteArrayList:**

* A concurrent alternative of synchronized List. CopyOnWriteArrayList provides better concurrency than synchronized List by allowing multiple concurrent readers and replacing the whole list on write operation.
* CopyOnWriteArraySet is a concurrent replacement to Synchronized Set.

**3.BlockingQueue:**

it easy to implement producer-consumer design pattern by providing inbuilt blocking support for the put() and take() method. put() method will block if the Queue is full while the take() method will block if the Queue is empty.

**4.PriorityBlockingQueue:**

another implementation of BlockingQueue which is ordered on priority and useful if you want to process elements on order other than FIFO.

**5. Deque and BlockingDeque:**

* Deque interface is added in Java 6 and it extends Queue interface to support insertion and removal from both ends of Queue referred to as head and tail.
* concurrent implementation of Deque like ArrayDeque and LinkedBlockingDeque.

**6. ConcurrentSkipListMap and ConcurrentSkipListSet:**

* Just like ConcurrentHashMap provides a concurrent alternative of synchronized HashMap.
* ConcurrentSkipListMap == TreeMap and ConcurrentSkipListSet == TreeSet
* They are sorted by natural order.
* For example instead of using TreeMap or TreeSet wrapped inside synchronized Collection, You can consider using ConcurrentSkipListMap or ConcurrentSkipListSet from java.util.concurrent package.

**Fail fast and fail safe:**

[**https://chatgpt.com/c/6735c4e8-4384-8010-8789-134241f9cedd**](https://chatgpt.com/c/6735c4e8-4384-8010-8789-134241f9cedd)

* **A fail-fast iterator** will immediately throw an exception (typically a ConcurrentModificationException) if the collection is modified while it is being iterated.
* The idea is that the program will "fail fast" rather than continuing in an unpredictable or incorrect state.
* Behavior: If you modify a collection while iterating over it (for example, by adding or removing elements), a ConcurrentModificationException will be thrown.
* Why fail-fast? The goal is to detect and report errors as soon as possible to avoid corrupting the data or introducing subtle bugs.
* Example: Most standard Java collections (like ArrayList, HashSet, etc.) use fail-fast iterators.
* **A fail-safe iterator**, on the other hand, works by making a copy of the collection or using a separate internal structure for iteration. If the collection is modified during iteration, the iterator will continue using the original structure and will not throw exceptions. It is designed to allow the collection to be modified safely while iterating.
* Behavior: Fail-safe iterators do not throw exceptions when the collection is modified during iteration. Instead, they may return unexpected results or behavior.
* Why fail-safe? The goal is to allow iteration to continue, even if the collection is modified. However, the downside is that modifications made during iteration may not be reflected in the iteration process.
* Example: Collections like CopyOnWriteArrayList are fail-safe. They handle concurrent modifications by creating a new copy of the collection when modified.

**JAVA 8: –**

**What new features did Java 8 introduce?**

The latest version has:

* An improved, immutable JodaTime-inspired **Date and time API.**
* **Functional Interfaces**
* A new language called **Lambda Expressions** that treats actions as objects.
* **Method References**, which enable defining Lambda Expressions by referring to methods directly using their names.
* **Default methods**, which give users the ability to add full implementations in interfaces besides abstract methods.
* **Stream API**, a special iterator class that allows processing object collections in a functional manner.

**Need for new version?**

* Dramatic changes in hardware created the need for Java to use current multi-core CPUs more efficiently.
* Enable users to use new Functional Programming (FP) features.

**Advantages of Java 8?**

* More concise, readable, testable, maintainable, scalable, productive and reusable code.
* More concurrent

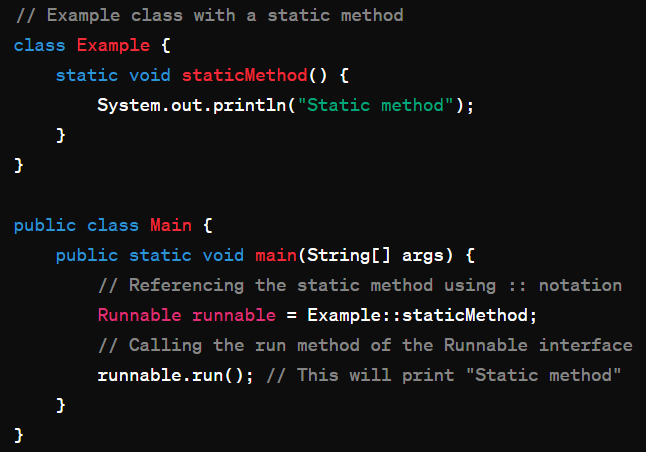
**Method References:**

method references provide a concise way to refer to methods or constructors. They can often replace lambda expressions when those lambda expressions are simply calling a method.

**1.Static method Reference:**

**ClassName :: methodName -** for static method

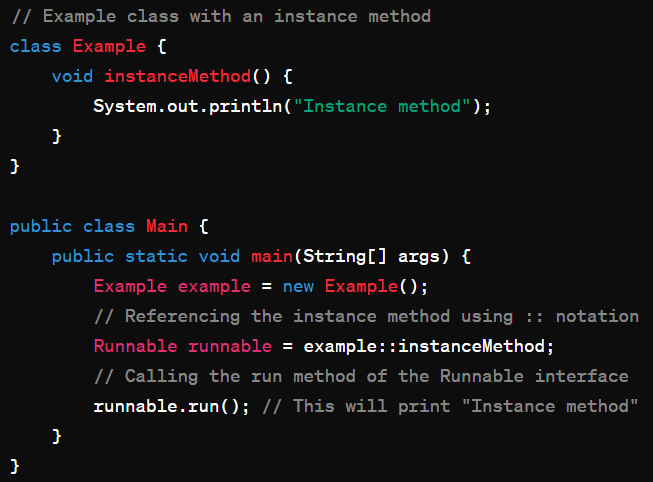
Ex:



**2.Instance method reference:**

**objectRef:: methodName -** for Instance method

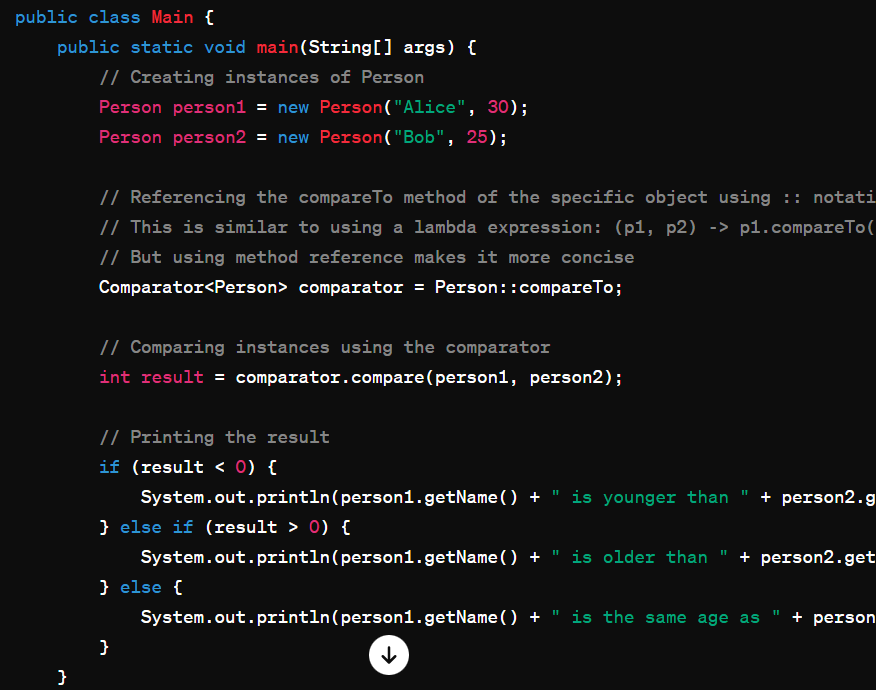
Ex:



**3.Instance Method on an Existing Object default method:**

**objectRef :: methodName -** for Instance method of a specific object or belongs to generic class like comparTo().

Ex:

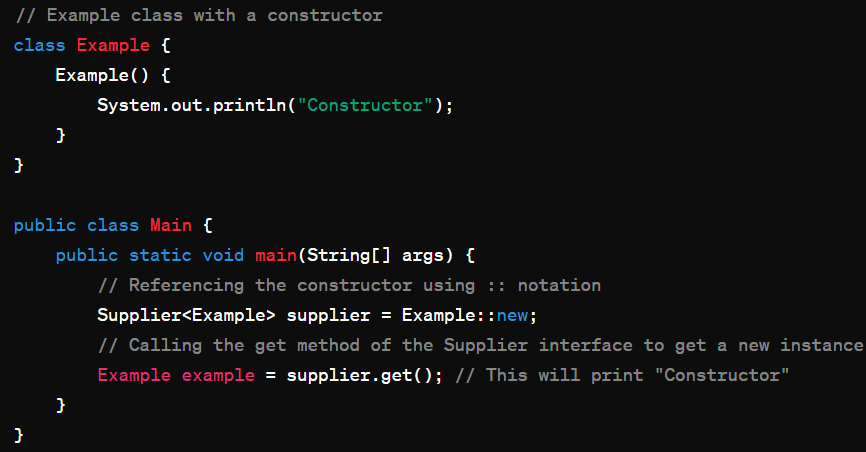


**4.Constructor Reference:**

**ClassName :: new -** for Constructor

No need to pass arguments to method references.

Ex:



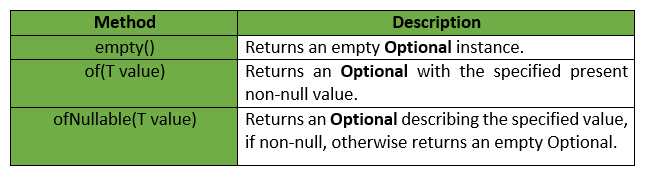
**Optional Class:**

* It is a container object which may or may not have non-null values and used to avoid need for multiple null checks and nullpointerException.
* Present in **java.util package**

This is a ***value-based*** class, i.e. their instances are :

* Final and immutable (though may contain references to mutable objects).
* Considered equal solely based on equals(), not based on reference equality(==).
* Do not have accessible constructors.

Static Methods in Optional Class:



isPresent(): Returns true if the value is present, otherwise false.

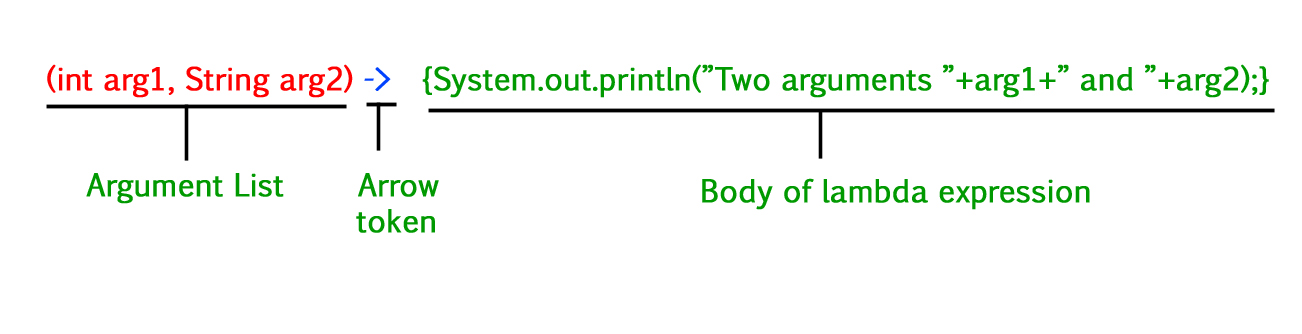
get(): Returns the value if present; otherwise, throws NoSuchElementException.

orElse(T other), orElseGet(Supplier<? Extends T> supplier), orElseThrow(Supplier<? Extends X> exceptionSupplier)

<https://chatgpt.com/share/674755be-3a5c-8010-a90b-9c54edb21b5f>

**Lambda Expressions:**

* They express instances of functional interfaces (An interface with a single abstract method is called a functional interface).
* Lambda Expressions are anonymous functions.
* Enable to treat functionality as a method argument, or code as data.
* A function that can be created without belonging to any class.
* A lambda expression can be passed around as if it was an object and executed on demand.
* Syntax :



There are three Lambda Expression Parameters are mentioned below:

1. Zero Parameter
2. Single Parameter
3. Multiple Parameters

**Functional Interfaces:**

1. Function – apply() - Takes one argument and returns a result
   * andThen() – executes after current function
   * compose() - executes before current function
   * identity() - returns its own argument
2. Consumer – accept() - Takes one argument and returns no result

- andThen() – executes after accept()

Ex: add.andThen(display).accept(5,6);

1. Supplier – get() - Takes no argument and returns a result

- andThen() – executes after current function

4. Predicate – test() - Takes one argument and returns a Boolean

- isEqual() -checks if two objects are equal.

- and() – a short-circuit logical AND of two predicates

- negate() – logical negation

- or() – a short-circuit logical OR of two predicates

### **What is a default method, and when does it get used?**

* The default method involves an implementation, and it is found in the interface.
* The method adds new functionalities to an interface while preserving backward compatibility with the classes that already implement the interface.
* The default methods were introduced to provide backward compatibility so that existing interfaces can use the lambda expressions without implementing the methods in the implementation class.
* Default methods are also known as **defender methods**or **virtual extension methods**.

**Streams:**

* A ***stream is a sequence of objects that supports various methods*** which can be pipelined to produce the desired result.
* *If we want to represent a group of objects as a single entity then we should go for*[***collection***](https://www.geeksforgeeks.org/collections-in-java-2/)*.*
* *But if we want to process objects from the collection then we should go for streams.*
* Stream is available as an interface. Java.util.stream.

Ex: Stream s = c.stream();

* Components of Stream:

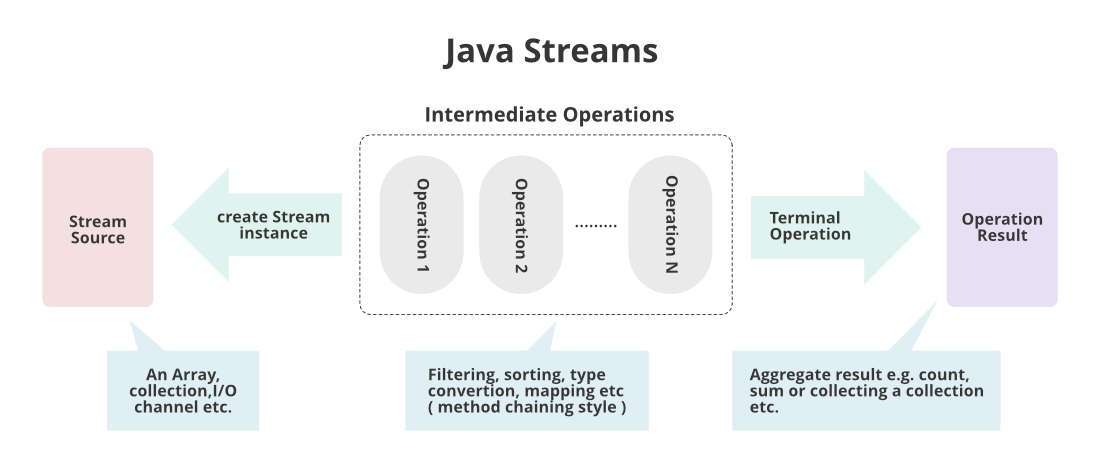
1.Sequence of Elements

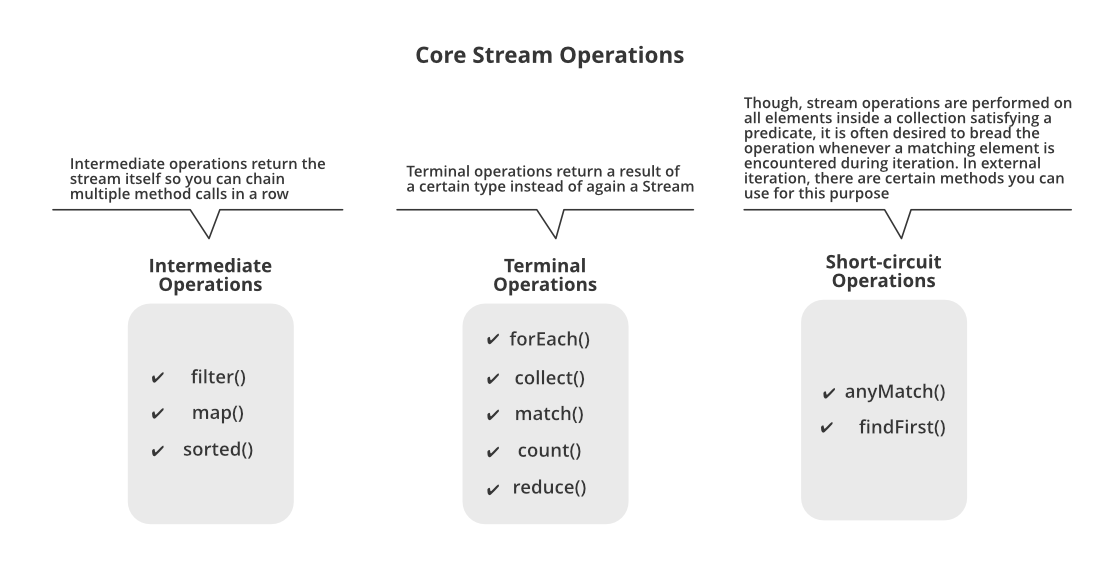
2.Source

3.Aggregate Operations

4.Pipelining

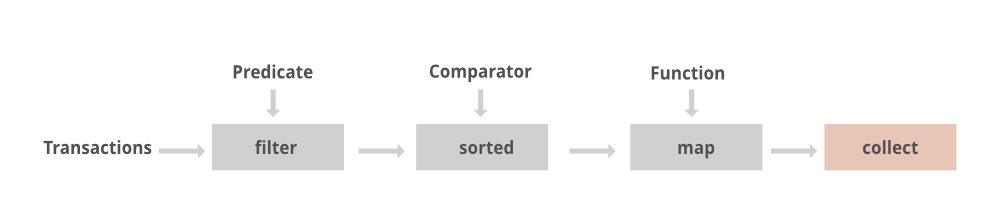
5.Internal iteration





There are 3 types of stream operations:

* 1. Intermediate operations – a stream can have many of this.
  2. Terminal operations – a stream can have only one.
  3. Short – circuit operations



Streams can be created in three ways:

1. Using an object of any class from the collection framework
2. Using an array of the reference data type
3. Using the interface defined in the ‘java.util.stream’ package

filter() – takes a predicate i.e. takes an argument of any type and returns Boolean. Uses predicate.

sorted() – returns a stream sorted in natural order.

Ex: stream.sorted(Comparator.reverseOrder()).forEach(System.out::println);

distinct() -returns stream of distinct elements.

map() – applies a function to each element in stream. Uses function

forEach() – uses consumer.

**Multithreading:**

* Multithreading is a Java feature that allows concurrent execution of multiple threads within a Java program and better utilization of system resources.
* A thread is the smallest unit of execution, representing an independent flow of control.
* Threads are lightweight processes within a process.
* In its lifetime, a thread undergoes the following states, namely:

1. New State:

When code is not run yet and execution process is not yet initiated.

1. Active State:

When start() method is invoked and this state has 2 sub states:

* 1. Runnable state:

Thread is ready to run.

* 1. Running state:

When thread receives CPU allocated by Thread Scheduler.

1. Waiting/Blocked State:

* When a thread is inactive but on a temporary time due to lack of resources to complete the task.
* Thread scheduler clears the threads in blocked or waiting state by rejecting unwanted threads and allocating CPU on priority basis.

1. Timed Waiting State:

In order to avoid threads from waiting for a long time, each thread has a time period for which sleep() method is invoked and after the time expires the thread starts executing its task.

1. Terminated State:

When thread

* Finishes its task.
* Faces segmentation faults, exceptions etc. (Abnormal Termination)
* Is dead or no longer available.

We can create Threads in java using two ways, namely :

1. Extending Thread Class:

* Involves overriding the run() method.
* We create a class that extends the **java.lang.Thread** class.
* Just use the class name to create thread and give t.start().

We use the following constructors for creating the Thread:

* Thread
* Thread(Runnable r)
* Thread(String name)
* Thread(Runnable r, String name)

Ex:

ThreadP1 t = **new** ThreadP1();

t.start();

1. Implementing a Runnable interface:

* We create a new class which implements java.lang.Runnable interface and override run() method.
* Create a class object and use that to create thread by feeding it to thread class object and give t.start();

Ex:

ThreadP2 rt = **new** ThreadP2();

Thread t = **new** Thread(rt);

t.start();

Thread Class Vs. Runnable Interface:

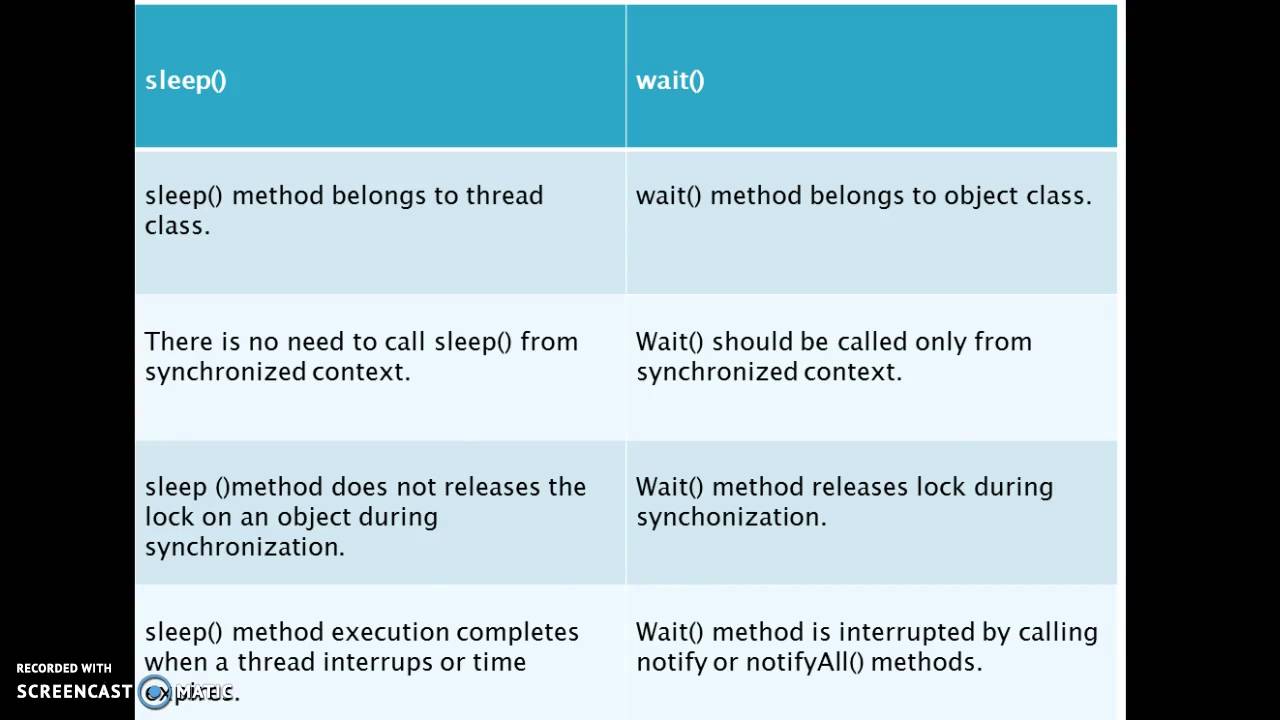
* If we extend Thread class, our class cannot extend any other class whereas by implementing Runnable our class can extend other classes.
* We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.
* Using runnable will give you an object that can be shared amongst multiple threads.

Start() Vs. run():

* When a program calls the [*start()* method](https://www.geeksforgeeks.org/start-function-multithreading-java/), a new thread is created and then the *run()* method is executed. But if we directly call the *run()* method then no new thread will be created and *run()* method will be executed as a normal method call on the current calling thread itself and no multi-threading will take place.
* we can’t call the *start()* method twice otherwise it will throw an[*IllegalStateException*](https://www.geeksforgeeks.org/how-to-solve-java-lang-illegalstateexception-in-java-main-thread/)whereas *run()* method can be called multiple times as it is just a normal method calling.

Sleep():

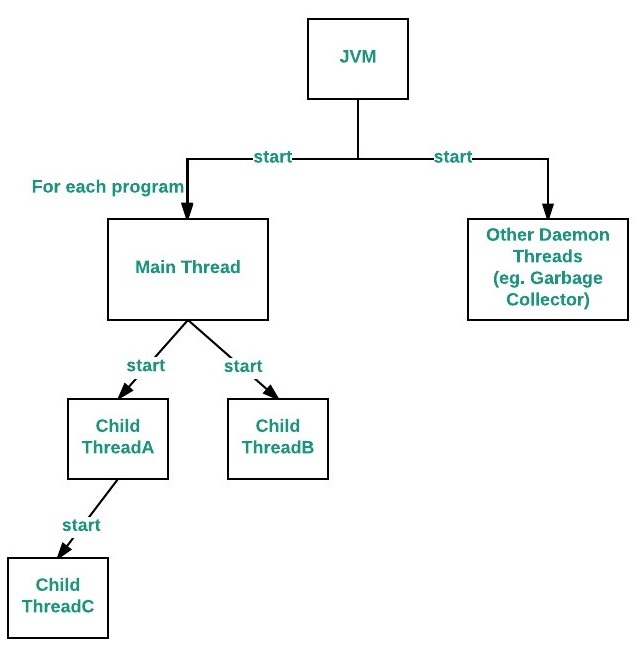
* Thread class contains the **Sleep()**method.
* If any other thread interrupts when the thread is sleeping, then InterruptedException will be thrown.
* It has parameters: sleep(long millis, int nanos).
* Return type of sleep() is void and is static



**Thread Priority:**

* Threads in Java can have a priority ranging from Thread.MIN\_PRIORITY to Thread.MAX\_PRIORITY.
* The priority determines the order in which threads are scheduled to run. Higher-priority threads get more CPU time.
* The default priority is set to 5 as excepted.
* Minimum priority is set to 1.
* Maximum priority is set to 10.

**Main Thread:**



* we create Main Method in each and every Java Program, which acts as an entry point for the code to get executed by JVM.
* Similarly in this Multithreading Concept, Each Program has one Main Thread which was provided by default by JVM.
* Hence whenever a program is being created in java, JVM provides the Main Thread for its Execution.
* It is the thread from which other “child” threads will be spawned.
* Often, it must be the last thread to finish execution because it performs various shutdown actions.

Daemon thread:

* It is a low-priority thread that performs background operations such as garbage collection, finalizer, Action Listeners, Signal dispatches, etc.
* Daemon thread in Java is also a service provider thread that helps the user thread.

Reentrant Lock:

* A synchronization primitive provided by Java's concurrency utilities (java.util.concurrent.locks) that enables exclusive access to a shared resource by multiple threads.
* It's called "reentrant" because it allows a thread to acquire the lock multiple times without deadlocking itself, as long as it releases the lock an equal number of times.

Here's how a ReentrantLock works:

1. **Acquiring the Lock:**

* When a thread wants to access a critical section of code or a shared resource protected by a ReentrantLock, it attempts to acquire the lock using the lock() method.
* If the lock is available, the thread acquires it and proceeds to execute the critical section. If the lock is not available because it's held by another thread, the current thread will block until the lock becomes available.

1. **Reentrant Behavior:**

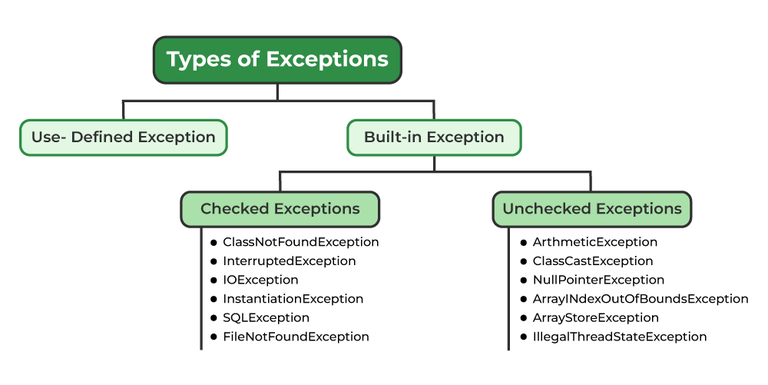
* If a thread that already holds the lock attempts to acquire it again (nested locking), the lock will be granted without blocking the thread.
* The thread must release the lock an equal number of times it acquired it.

1. **Releasing the Lock:**

* After the thread finishes executing the critical section or accessing the shared resource, it must release the lock using the unlock() method.
* Releasing the lock allows other threads that are waiting to acquire the lock to proceed.
* ReentrantLock provides more flexibility and functionality compared to synchronized blocks/methods, such as timed lock acquisition, interruptible lock acquisition, and non-blocking lock acquisition (tryLock()).
* It's particularly useful in scenarios where finer-grained control over locking behavior is needed.

**Exception Handling:**





**Exception:**

* It is an event that disturbs or stops the normal execution of a program.

**Throw & Throws:**

* The throw statement is used to explicitly throw an exception within a method or block of code.
* It is followed by an instance of an exception or a throwable object.
* **Throw should be the last statement of a block.**

Ex:

throw new SomeException("This is an example exception");

* The **throws** clause is used in method declarations to indicate that the method might throw certain types of exceptions.
* It is followed by a list of exception classes that the method may throw.

Ex:

public void exampleMethod() throws SomeException, AnotherException {

// method implementation

}

* When a method is declared with a **throws** clause, it means that the method can throw exceptions of the specified types, and calling code that invokes this method must either catch these exceptions or declare that it also throws them.
* The calling method must either handle the exception using a **try-catch** block or declare that it also **throws Exception**.
* If the current method is not equipped to handle an exception, it can propagate the exception up the call stack by declaring it with **throws**. This allows higher-level methods to handle the exception.
* If the exception is a runtime exception (subclass of **RuntimeException**), you are not required to declare it in the **throws** clause.
* However, it is still a good practice to document the potential exceptions that a method may throw, even if they are unchecked. This helps in understanding the expected behavior of the method.

Here's a simple example illustrating the use of **throw** and **throws**:



**Try, Catch & Finally:**

Try{

}catch(Exception e){

System.out.println(e.toString());

}finally{

//this will always get executed

}

**Try with Resources** **(Java 7 and later):**

try(declare resources here) {

// use resources

}

catch(FileNotFoundException e) {

// exception handling

}

* Try is always mandatory
* Catch is not mandatory if finally is present.
* If an exception is thrown in both try block and finally block, the method returns the exception thrown in finally block.
* Try with Resources is a try statement that declares one or more resources in it.
* The try-with-resources statement ensures that each resource is closed at the end of the statement execution.
* You can pass any object as a resource that implements java*.*lang*.*AutoCloseable.
* You cannot use parent and child exceptions in same catch. Should use exceptions of same level.

Ex:

try {

str.charAt(10/0);

} catch (IndexOutOfBoundsException | ArithmeticException e) {

System.out.println(e);

}

**Final Keyword:**

The final keyword in Java is used as a modifier for various entities such as classes, methods, and variables. Its meaning and implications depend on where it is applied:

**Final Variable:**

When applied to a variable, it means that the variable's value cannot be changed once it has been assigned.

Ex:

final int x = 10; // x cannot be reassigned

**Final Method:**

When applied to a method, it means that the method cannot be overridden by subclasses.

Ex:

class Parent {

final void display() {

System.out.println("This method cannot be overridden.");

}

}

**Final Class:**

When applied to a class, it means that the class cannot be subclassed.

Can make an object or class immutable using final and removing the setters.

Ex:

final class FinalClass {

// Class implementation }

**Final Parameter:**

When applied to a method parameter, it means that the parameter cannot be modified within the method.

Ex:

void process(final int value) {

// value cannot be modified within this method

}

The final keyword contributes to code safety, immutability, and design integrity by preventing certain modifications or extensions.

**Finalize Method:**

* The finalize method is a method defined in the Object class, which is the root class for all Java classes.
* It is called by the garbage collector before an object is garbage collected.
* The purpose of the finalize method is to allow an object to perform cleanup operations before it is reclaimed by the garbage collector.

Ex:

class MyClass {

// Other class members

@Override

protected void finalize() throws Throwable {

// Cleanup operations before garbage collection

super.finalize();

}

}

* However, it's important to note that relying on finalize for critical resource cleanup is discouraged in modern Java development.
* Instead, developers are encouraged to use other mechanisms, such as the AutoCloseable interface and try-with-resources statement, to ensure proper resource management.

In summary:

* final is a keyword used for different purposes to denote immutability, prevention of inheritance, or prevention of method overriding.
* finalize is a method defined in the Object class, which allows objects to perform cleanup operations before being garbage collected.