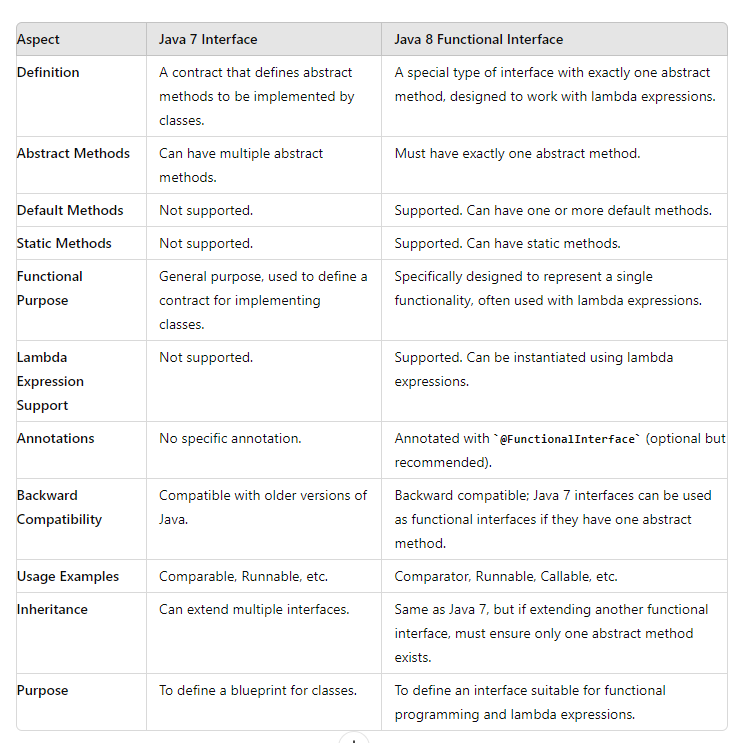
**Does Interface and functional interface are same** ?

No, interfaces and functional interfaces are not the same, although functional interfaces are a subset of interfaces in Java.

Here's the distinction: **Functional interfaces** have a specific constraint of containing **only one abstract method**, while **regular interfaces in Java can contain any number of abstract methods,** default methods, and static methods (with Impl).

We cannot Create a Object for Java7 Interface, as it doesn’t contain any Impl, whereas for **Functional Interface can be Instantiated(Object Creation) using Lambda expressions or method reference**

****

**Summary**

* **Java 7 Interface**: Allows multiple abstract methods, doesn't support default/static methods, and doesn't work with lambda expressions.
* **Java 8 Functional Interface**: Designed for functional programming with a single abstract method, supports default and static methods, and works seamlessly with lambda expressions.

**Functional Interface :**

An Interface that contains exactly **one abstract method** is known as functional interface. It can **have any number of default, static methods(with impl)** but can contain only one abstract method.

Functional Interface is also known as **Single Abstract Method** Interfaces or SAM Interfaces. It is a new feature in Java, which helps to achieve functional programming approach.

**@FunctionalInterface** annotation is added so that we can mark an interface as functional interface.

It is not mandatory to use it, but it’s best practice to use it with functional interfaces to avoid addition of extra abstract methods accidentally. If the interface is annotated with @FunctionalInterface annotation and we try to have more than one abstract method, it throws compiler error.

Functional Interface facilitate the use of lambda expressions and method references, which in turn enhance code readability, support functional programming paradigms, and enable features like the Streams API for efficient collection processing. They represent a fundamental aspect of modern Java programming, promoting concise, readable, and maintainable code.

Default Methods in Functional Interface are mainly introduced to solve the problem (Backward Compatibility meaning a new feature or version should support older version without breaking) like, if we want to add a new feature or method in an interface then we should add that as abstract method in interface then all the child classes need to break the existing contract and need to provide implementation, instead of doing that if we are able to add implementation directly in interface then we don’t need to break or modify child class code. during java 8 development when they tried to add new feature in collection framework, they had this issue so they introduced this feature.

By using Default Methods in Functional Interfaces Child classes no need to break existing Implementation, if we want can use Functional Interface default method or else if we require then we can override default method Impl ,

Static methods in Functional interface are mainly used for some common or utility purpose if we have some common impl which require for all child classes we can use static methods as we cannot override static methods.

 Functional interfaces can have static and final variables.

 Functional interfaces cannot have constructors or instance variables.

 You cannot directly instantiate a functional interface, but you can create objects using lambda expressions or anonymous classes.

**Some Built-in Java Functional Interfaces**

Since Java SE 1.8 onwards, there are many interfaces that are converted into functional interface. All these interfaces are annotated with @FunctionalInterface. These interfaces are as follows –

* **Runnable –>** This interface only contains the run() method.
* **Comparable –>** This interface only contains the compareTo() method.
* **ActionListener –>** This interface only contains the actionPerformed() method.
* **Callable –>** This interface only contains the call() method.

**Java SE 8 included four main kinds of functional interfaces in java.util.function package**which can be applied in multiple situations. These are:

1. Consumer
2. Predicate
3. Function
4. Supplier

Amidst the previous four interfaces, the first three interfaces,i.e., Consumer, Predicate, and Function, likewise have additions that are provided beneath –

1. Consumer -> Bi-Consumer
2. Predicate -> Bi-Predicate
3. Function -> Bi-Function, Unary Operator, Binary Operator

**Lambda expressions**

Lambda expressions enable Java’s object-oriented programming world with functional programming. It **provides an implementation of an interface which has a functional interface**. When lambda functions are used, we don’t need to define the method by providing the implementation. In fact, lambda expressions are treated as a function.

The basic syntax of Lambda expression is **(argument) → (body)**

In General Object oriented program every thing is represented as object only we cannot do anything with out object, it means if you want to print a simple hello word also we need to create a class , object and then hello world method. To avoid this in java 8 they introduced functional programming we can achieve this with functional interface and lambda expression

Example : if we want to print a helloword without functional interface we need to create a Greet interface with printGreet() abstract method then we need to provide implementation for printGreet() method in our own class like GreetImpl then create a object with that reference we call printGreet() method . But if we use functional interface we can just provide implementation for that without creating object or class

Greet g=() -> System.out.println(“hello word greet message”) g.printgreet(); Here we are providing implementation with lambda expression ()-> lambda expression or lambda function is a anonymous with without any method name, lambda expression must be a type of functional interface, it means as we assign some value to variable name of type int or String and we can pass int or string values as argument we can do same thing with lambda expression we can assign a function (lambda expression) to a variable of type FunctionalInterface and we can pass functional interface as argument but receiver method must have parameter type as functional interface.

**Functional programming reference** : [**https://www.geeksforgeeks.org/functional-programming-in-java-with-examples/**](https://www.geeksforgeeks.org/functional-programming-in-java-with-examples/)

Public **interface** Funtest{

**void** m2(String s,**int** x);

}

**Example** 1: varible f holding the function below Funtest f=(s,a)-> System.***out***.println(s+" "+a );

f.m2("hello world",10);

**Example** 2: passing function as argument

Public **class** Test{

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

*printmethod*((s,a)-> System.***out***.println(s+" "+a ));

}

**static** **void** printmethod(Funtest fun){

fun.m2("hello world",10);

}

}

Example 3:

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

Funtest f=(s)-> s.length();

System.***out***.println(f.m2("hello world"));

}

**public** **interface** Funtest {

**int** m2(String s);

}

Basic syntax is ()-> logic // for single line code For multiple line code : () -> { logic/////// } above two are without parameters we can have parameters and return type also for void method we don’t need to mention any return type example (a,b)-> a+b; // here a+b is returns value of type int in functional interface abstract method must be like **public int cal (int a,int b); another example for lambda expression**

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

Funtest f= (a,b)-> {

System.***out***.println(a+b);

System.***out***.println(a\*b);

**int** z=a+b;

**return** z;

};

System.***out***.println(f.cal(10, 20));

}

When we use single parameters for lambda expression we don’t even required to specify () bracket symbol also

**Example Funtest f=s -> s.length();**

**System.*out*.println(f.t("hello world"));**

int t(String s);// we should add this method in interface

point to remember in lambda expression left side will be parameter of input and right side to -> arrow symbol will be return type value

most important thing is a lambda expression must be assigned to a functional interface type variable or passed to argument to type of functional interface.

**Functional interfaces**

1. Consumer : Accepts **a single argument** and returns **no result --void** accept(T t); andThen
2. Predicate : which **accepts an argument** and **returns a boolean**. Usually, it used to apply in a filter for a collection of objects.-- **boolean** test(T t); and or negate
3. Function : Accepts **one argument** and **returns a result. --** R apply(T t);
4. Supplier :  it takes **no arguments** and returns **a result. --**T get();

**Convert String to Integer (below code same with Lamda and method ref)**

Function<String, Integer> stoi=(String s)->Integer.*valueOf*(s); //with Lambda

Function<String, Integer> stoix=Integer::*valueOf*; //with Method ref

stoix.apply("20");

System.***out***.println(stoi.apply("10"));

# **Java Method References**

Java provides a new feature called method reference in Java 8. **Method reference is used to refer method of functional interface**. It is **compact and easy form of lambda expression**. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference

## **Types of Method References**

There are following types of method references in java:

We can call a method by using :: symbol if we want to call static method we need to use class name , if we want to call instance method use object reference and if we want to call constructor use class name and new keyword, Note: for these method reference concept we must use functional interface , method reference will be reference of functional interface , method reference is like lamda expression.

1. **Reference to a static method**: This type of method reference refers to a static method. It is denoted by the syntax **ContainingClass::staticMethodName**.
2. **Reference to an instance method of a particular object**: This type of method reference refers to an instance method of a particular object. It is denoted by the syntax **objectReference::instanceMethodName**.
3. **Reference to an instance method of an arbitrary object(Specific Object) of a particular type**: This type of method reference refers to an instance method of an arbitrary object of a particular type. It is denoted by the syntax **ContainingType::instanceMethodName**.
4. **Reference to a constructor**: This type of method reference refers to a constructor. It is denoted by the syntax **ClassName::new**.

**Note: Main difference between 2nd and 3rd point is 2nd point refers to particular object like Ex: Person P= new Persion(“Teja”) // P::getName (only one obj type)**

**Where as 3rd point refers to Arrays.asList(new Persion(“Teja”), new Persion(“raja”)).stream().map(Person::getName) here person Type but each new object as different.**

**Syntax**: Classname::staticMethodName

Consumer<String> z= System.***out***::println;

z.accept("hello world");

**Syntax**:  ObjectReference::saySomething;

Main(){

Consumer<String> ss=tt::dummy;

ss.accept("hello dummy");

}

**public** **void** dummy(String x) {

System.***out***.println(x);

}

**Syntax**: ClassName::**new**  .

Tester(String x){

System.***out***.println(x +"inside constr");

}

Consumer<String> st=Tester::**new**;

st.accept("hello dummy call const");

}

**Syntax for lambda expression**

Consumer<String> z= x-> System.***out***.println(x)

z.accept("hello world");

**Below is Example of Reference to an instance method of an arbitrary object of a particular type**

List<Person> people = Arrays.asList(**new Person**("Alice"),**new Person**("Bob"),new Person("Charlie")); // Method reference to an instance method of an arbitrary object of a particular type

people.stream().map(**Person**::getName).forEach(System.out::println);

// Here, Person::getName is a method reference

# **Java Optional Class**

The **Optional** class in Java 8 is a **container object** that may or may not contain a non-null value. It was introduced to address the problem of null references and to provide a more explicit and safer way to represent the absence of a value.

Here are some key points about the **Optional** class: **Optional class to handle situations where a value may or may not be present.** Instead of using null, Optional provides a way to represent "no value" in a more meaningful and expressive way.

1. **Encourages explicit handling of null**: **Optional** encourages developers to handle null values explicitly by providing methods for checking the presence of a value (**isPresent()**), accessing the value safely (**get()**), or providing a default value if the value is absent (**orElse()**).
2. **Avoids NullPointerExceptions (NPEs)**: By using **Optional** instead of returning null from methods or dereferencing variables, developers can reduce the likelihood of encountering NPEs, as the need to explicitly check for null is enforced by the API.
3. **Promotes clearer and safer code**: Code using **Optional** tends to be clearer and safer because it explicitly communicates whether a value is present or absent, reducing the risk of accidentally dereferencing null values without checking for null.
4. **Not a replacement for all null occurrences**: It's important to use **Optional** judiciously and not as a blanket replacement for all occurrences of null. It's generally recommended to use **Optional** for return values of methods or parameters that may be absent, rather than as fields in classes or elements in collections.

Overall, the **Optional** class in Java 8 provides a more structured and safer way to handle the absence of values, leading to more robust and null-safe code.

Java introduced a new class Optional in jdk8. It is a public final class and used **to deal with NullPointerException in Java application**. You must import java.util package to use this class. It provides methods which are **used to check the presence of value for particular variable. IsPresent , get , filter are mostly used methods**

**In Optional class we mainly use** Optional.of() and Optional.ofNullable() methods we use Optional.of() when we are sure its not null like Optional.of(getResult) we use if present method to check if its present we handle success or else failure case in case if it is null Optional.of(null) it will throw null pointer exception, we use Optional.ofNullable(getResult) when we are not sure about value is null or not if incase value is null Optional.ofNullable(null) it will give us empty optional object, if don’t want to throw any error just return empty result we use this

Optional<String> s=Optional.*of*("").filter(x->x.startsWith("a"));

**if**(s.isPresent()) {

System.***out***.println("ok");

}

**else** {

System.***out***.println("not ok");

}

1. **of(T value)**:
   * Creates an **Optional** instance containing the specified non-null value.
   * Throws a **NullPointerException** if the provided value is null.
2. **ofNullable(T value)**:
   * Creates an **Optional** instance containing the specified value if it is non-null.
   * If the provided value is null, creates an empty **Optional** instance.
3. **empty()**:
   * Returns an empty **Optional** instance.
4. **get()**:
   * Returns the value if present, otherwise throws a **NoSuchElementException**.
   * It's recommended to avoid using **get()** and prefer other methods like **orElse()** or **orElseThrow()** to handle absent values safely.
5. **isPresent()**:
   * Returns true if a value is present, otherwise false.
6. **ifPresent(Consumer<? super T> consumer)**:
   * Performs the given action if a value is present.
7. **orElse(T other)**:
   * Returns the value if present, otherwise returns the specified default value.
8. **orElseGet(Supplier<? extends T> other)**:
   * Returns the value if present, otherwise calls the specified supplier to generate a default value.
9. **orElseThrow(Supplier<? extends X> exceptionSupplier)**:
   * Returns the value if present, otherwise throws an exception created by the provided supplier.
10. **map(Function<? super T,? extends U> mapper)**:
    * If a value is present, applies the given mapping function to it and returns the result wrapped in an **Optional**, otherwise returns an empty **Optional**.
11. **flatMap(Function<? super T,Optional<U>> mapper)**:
    * If a value is present, applies the given mapping function to it and returns the result, otherwise returns an empty **Optional**.

[**https://www.javatpoint.com/java-8-optional**](https://www.javatpoint.com/java-8-optional)

# **Java forEach loop**

Java provides a new method forEach() to iterate the elements. It is defined in Iterable and Stream interface. It is a default method defined in the Iterable interface. Collection classes which extends Iterable interface can use forEach loop to iterate elements.

This method takes a single parameter which is a functional interface. So, you can pass lambda expression as an argument.

1. **default** **void** forEach(Consumer<**super** T>action)
2. List<String> gamesList = **new** ArrayList<String>();
3. gamesList.add("Football");
4. gamesList.add("Cricket");
5. gamesList.add("Chess");
6. gamesList.add("Hocky");
7. System.out.println("------------Iterating by passing method reference---------------");
8. gamesList.forEach(System.out::println);

// gamesList.forEach(games -> System.out.println(games));

**forEachOrdered().** It is used to iterate elements in the order specified by the stream.

# **Default Methods** **In Java 8**

Before Java 8, interfaces could have only abstract methods. The implementation of these methods has to be provided in a separate class. So, if a new method is to be added in an interface, then its implementation code has to be provided in the class implementing the same interface. To overcome this issue, Java 8 has introduced the concept of default methods which allow the interfaces to have methods with implementation without affecting the classes that 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**.

**Important Points:**

1. Interfaces can have default methods with implementation in Java 8 on later.
2. Interfaces can have static methods as well, similar to static methods in classes.
3. Default methods were introduced to provide backward compatibility for old interfaces so that they can have new methods without affecting existing code.

**For example**, ‘List’ or ‘Collection’ interfaces do not have ‘forEach’ method declaration. Thus, adding such method will simply break the collection framework implementations. Java 8 introduces default method so that List/Collection interface can have a default implementation of forEach method, and the class implementing these interfaces need not implement the same.

**Ambiguity Problem with Default Methods in Functional Interfaces and how to Solve them**

# When a class implements two interfaces that both have default methods with the same signature, there is a conflict because the compiler cannot automatically determine which default method should be used in the implementing class. Java forces the developer to explicitly resolve this conflict by overriding the method in the implementing class.

class MyClass **implements MyInterfaceA, MyInterfaceB** {

@Override

public void myDefaultMethod() {

// Must resolve the conflict explicitly

MyInterfaceA.super.mydefaultMethod(); // or MyInterfaceB.super.mydefaultMethod();

}

}

# **Stream In Java**

Introduced in Java 8, the Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.  
The features of Java stream are –

* A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
* Streams don’t change the original data structure, they only provide the result as per the pipelined methods.
* Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

Different Operations On Streams-  
**Intermediate Operations:**

**Intermediate Operations**: Intermediate operations are operations that transform or manipulate the elements of a stream. **These operations are typically lazy, meaning they do not process the elements of the stream until a terminal operation is invoked.** Intermediate operations are chainable, meaning you can apply multiple intermediate operations sequentially on a stream. Some common intermediate

1. **map:**map will transform one input to another form example if we have Stream of strings we want to know length of each string in stream we use **Array.asList(“teja”,”naik”).stream().map(String::length). .collect(Collectors.toList())**

The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.  
List number = Arrays.asList(2,3,4,5);  
List square = number.stream().map(x->x\*x).collect(Collectors.toList());

1. **flatMap**

 **Purpose**: Used to transform each element of a stream into a new stream, and then flatten all those individual streams into a single stream of values.

 **Use Case**: When you have nested lists or collections and you want to merge them into a single collection.

1. **filter:** Filter will gives us required result based on condition (filter data from stream as per our requirement ) like filter Strings only starts with “S”

**Arrays.asList("sam","dam").stream().filter(x->x.startsWith("s")) .collect(Collectors.toList());**

The filter method is used to select elements as per the Predicate passed as argument.  
List names = Arrays.asList("Reflection","Collection","Stream");  
List result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());

1. **sorted:** The sorted method is used to sort the stream.  
   List names = Arrays.asList("Reflection","Collection","Stream");  
   List result = names.stream().sorted().collect(Collectors.toList());
2. **distinct**: Returns a stream by removing duplicates(only unique elements will be present). Use Case: When you want to remove duplicate elements from a stream.
3. **Limit:** it will keep only specified elements example limit(3) it will return only first 3 elements
4. **Skip**: it will skip(remove) specified elements example skip(2) it will remove starting 2 elements
5. **peek** : Used to perform an action on each element of a stream as it is being processed, usually for debugging or logging purposes. It does not modify the stream. Use Case: Debugging a stream pipeline by printing intermediate values.

// Create a stream of integers **from 1 to 10**

Stream<Integer> stream = Stream.iterate(1, n -> n + 1).**limit(10);**

**// Skip the first 5 elements** of the stream and print the rest

**stream.skip(5).**forEach(System.out::println);

**Terminal Operations:**

Terminal operations are operations that produce a result or side-effect from a stream. When a terminal operation is applied to a stream, it triggers the execution of the entire stream pipeline, causing the intermediate operations to be evaluated. After the terminal operation is executed, the stream is considered consumed and cannot be reused. Some common terminal operations

1. **collect:** The collect method is used to return the result of the intermediate operations performed on the stream.  
   List number = Arrays.asList(2,3,4,5,3);  
   Set square = number.stream().map(x->x\*x).collect(Collectors.toSet());
2. **forEach:** The forEach method is used to iterate through every element of the stream.  
   List number = Arrays.asList(2,3,4,5);  
   number.stream().map(x->x\*x).forEach(y->System.out.println(y));
3. **reduce:** The reduce method is used to reduce the elements of a stream to a single value.  
   The reduce method takes a BinaryOperator as a parameter.

List number = Arrays.asList(2,3,4,5);  
int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);

Here ans variable is assigned 0 as the initial value and i is added to it .

anyMatch, allMatch, nonmatch, findFirst, findany, count, min, max

**Important Points/Observations:**

1. A stream consists of source followed by zero or more intermediate methods combined together (pipelined) and a terminal method to process the objects obtained from the source as per the methods described.
2. Stream is used to compute elements as per the pipelined methods without altering the original value of the object.

# **Java Collectors**

Collectors is a final class that extends Object class. It provides reduction operations, such as accumulating elements into collections, summarizing elements according to various criteria, etc.

Java Collectors class provides various methods to deal with elements

<https://www.geeksforgeeks.org/java-collectors/>

**What is a Stream?**

In Java 8, a **Stream** is a sequence of elements that supports various operations to process those elements, such as filtering, mapping, and reducing. A stream can be created from collections, arrays, or I/O channels, and it operates on data in a declarative, functional style.

**Stream (Sequential Stream)**

* **Execution:** Executes tasks sequentially, meaning each element in the stream is processed one after the other in a single thread.
* **Use Case:** Best suited for scenarios where the tasks are small.
* **Order:** Maintains the order of elements as per the source.
* **Performance:** Generally better for smaller datasets

**Stream:**

* **Advantages**: Simple and straightforward. No thread-safety concerns if the underlying data is not modified during processing.
* **Drawbacks**: May be slower for large datasets or complex operations, as it only uses a single thread.
* **When to Use**: Small datasets, simple operations, or when maintaining the order of processing is important.

**What is a Parallel Stream?**

A **Parallel Stream** is a type of stream that divides the source of elements into multiple sub-streams, which are processed in parallel on different threads. This can significantly speed up the processing of large datasets by utilizing multiple CPU cores.

* **Execution:** Executes tasks in parallel by dividing the source into multiple sub-tasks, which are processed in multiple threads simultaneously.
* **Use Case:** Ideal for large datasets or computationally intensive operations where parallel processing can significantly reduce execution time.
* **Order:** May not maintain the order of elements unless explicitly requested (e.g., using forEachOrdered()).
* **Performance:** Can offer significant performance improvement for large datasets but can also introduce overhead due to thread management and potential synchronization issues.

List<String> list = Arrays.asList("a", "b", "c", "d"); **list.parallelStream().**forEach(System.out::println);

**Parallel Stream:**

* **Advantages:** Potentially faster for large datasets by leveraging multiple CPU cores. Can significantly reduce processing time for computationally intensive tasks.
* **Drawbacks:** Introduces overhead from thread management. May lead to thread-safety issues if operations modify shared data. The order of processing might not be preserved, which can lead to unexpected results if order matters.
* **When to Use:** Large datasets, operations that can be done independently in parallel, or when performance is critical and you have a multi-core CPU.

When working with large integer datasets using parallel streams, data consistency depends on how you handle the data within the stream operations.

**Key Points to Consider:**

1. **Immutable Data Operations:**
   * If the operations performed on the data are **stateless** and **non-interfering**, meaning they do not modify the original data or depend on other elements' state, then data consistency is generally ensured.
   * Example: Summing up all integers in a dataset is a stateless operation, and using a parallel stream will maintain consistency.

java

Copy code

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, ...);

int sum = numbers.parallelStream().mapToInt(Integer::intValue).sum();

1. **Stateful Operations:**
   * If the operations are **stateful** or **interfering**, meaning they rely on shared mutable state or modify elements during processing, data consistency issues can arise in a parallel stream.
   * Example: Modifying a shared variable within a parallel stream without proper synchronization can lead to inconsistent results.

java

Copy code

int[] sum = {0};

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, ...);

numbers.parallelStream().forEach(num -> sum[0] += num); // Not thread-safe!

1. **Thread Safety:**
   * To ensure data consistency with parallel streams, operations on shared resources must be thread-safe. This can be achieved by using synchronization, atomic classes (like AtomicInteger), or avoiding shared mutable state altogether.
   * Example: Using AtomicInteger to ensure thread safety.

java

Copy code

AtomicInteger sum = new AtomicInteger(0);

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, ...);

numbers.parallelStream().forEach(num -> sum.addAndGet(num)); // Thread-safe

1. **Collecting Results:**
   * When collecting results from parallel streams, using thread-safe collectors (like Collectors.toList() for immutable lists or Collectors.toConcurrentMap()) ensures data consistency.
   * Example: Collecting a list of unique integers.

java

Copy code

List<Integer> uniqueNumbers = numbers.parallelStream()

.distinct()

.collect(Collectors.toList());

**Conclusion**

* **Data consistency** in parallel streams is guaranteed if the operations are stateless, thread-safe, or involve immutable data structures.
* **Inconsistent results** can occur if stateful operations or shared mutable data are involved without proper synchronization.

If you follow best practices and ensure that operations are thread-safe or avoid shared mutable state, you can confidently use parallel streams with large datasets while maintaining data consistency.

**Synchronous and Asynchronous in Java**

In Java, synchronous and asynchronous operations refer to how tasks are executed relative to other tasks. The key difference lies in whether a task waits for another task to complete before proceeding.

1**. Synchronous Operations**

In a synchronous operation, tasks are executed sequentially. Each task must complete before the next one begins. The calling thread waits until the current task finishes.

**2. Asynchronous Operations**

In an asynchronous operation, tasks are executed independently of one another. The calling thread initiates a task and immediately proceeds without waiting for the task to complete. The completion of the task is handled separately, often through callbacks or futures.

**Example Scenario**

Imagine you're building a web server that processes multiple client requests. If each request involves calling an external service (like fetching data from a database or another API), using asynchronous calls allows the server to handle multiple requests concurrently:

* **Synchronous Approach:** The server can handle one request at a time, waiting for the external service to respond before moving on to the next request.
* **Asynchronous Approach:** The server can initiate the external service call for one request, move on to handle other requests, and process the responses as they arrive, leading to better utilization of server resources and improved throughput.

This asynchronous model is especially useful in I/O-bound operations where waiting for responses (e.g., network latency) can be time-consuming, allowing the system to remain responsive and handle more tasks concurrently.

**Why Use CompletableFuture When We Have Future?**

Feature was introduced in java 5 for Async programming, it doesn’t have non-blocking methods like thenApply, thenAccept and exceptionally like chaining and combining exceptionally features are not supported so in java 8 CompletableFeature introduced by impl Feature interface that supports chaining and combining exceptionally additionally

**Future**

In Java, the Future interface is a part of the java.util.concurrent package and provides a mechanism to retrieve the result of an asynchronous computation.

 **Future**: Represents the result of an asynchronous computation and provides methods to retrieve the result or handle exceptions.

 **FutureTask**: A concrete implementation of Future that wraps a Callable or Runnable task.

 **CompletableFuture**: Extends Future and provides additional capabilities for asynchronous programming, including chaining and handling exceptions.

 **ScheduledFuture**: Represents the result of a scheduled computation, such as delayed or periodic tasks.

**How Future Works in Asynchronous Programming**

1. **Submitting a Task:**

To start an asynchronous task, you submit a Callable or Runnable to an ExecutorService. The ExecutorService executes the task in a separate thread.

ExecutorService executor = Executors.newSingleThreadExecutor(); Future<Integer> future = executor.submit(() -> { // Task logic return 42; });

 **Task Execution:**

* The task is executed asynchronously in a separate thread managed by the ExecutorService. The main thread continues to execute other code without waiting for the task to complete.

** Getting the Result:**

* To retrieve the result of the asynchronous computation, you call the get() method on the Future object.
* The get() method is blocking, meaning it will wait until the computation is complete and the result is available.
* If the task has not completed, get() will block until the result is ready.

**Handling Exceptions:**

* If the task throws an exception during execution, get() will throw an ExecutionException.
* You can handle exceptions using try-catch blocks when calling get().

**Checking Status:**

* The Future interface provides methods to check the status of the task:
* isDone(): Returns true if the task is completed (whether successfully, due to cancellation, or with an exception).
* isCancelled(): Returns true if the task was cancelled before it completed.

**What is CompletableFuture?**

CompletableFuture is a class in Java that represents a future result of an asynchronous computation. It was introduced in Java 8 as part of the java.util.concurrent package. This class is used for handling asynchronous programming, where tasks are executed in the background, and their results are handled once they are complete.

**What Does CompletableFuture Do?**

CompletableFuture allows you to:

1. **Run Asynchronous Tasks**: Execute tasks in the background without blocking the main thread. **Asynchronous Execution**: You can execute tasks asynchronously using methods like runAsync() or **supplyAsync(),** which run tasks in a separate thread pool.
2. **Chain Tasks**: Chain multiple asynchronous operations together, where each task can start based on the result of the previous one using methods like **thenApply**, **thenAccept**, and **thenCompose**.
3. **Combine Results**: Combine the results of multiple asynchronous tasks. using methods like **thenCombine**, **thenAcceptBoth**, and **allOf**.
4. **Exception Handling**: It provides methods like exceptionally and handle to handle exceptions that occur during asynchronous execution.
5. **Non-Blocking Operations**: it allows you to work with the result of an asynchronous operation without blocking the main thread.

**How Threads Are Managed in CompletableFuture**

1. Creating a New Thread for the Asynchronous Task:

When you use CompletableFuture.supplyAsync() (or other similar methods), a new thread is indeed created to execute the task asynchronously. By default, this thread comes from the ForkJoinPool.commonPool(), which is a pool of worker threads managed by the Fork/Join framework. 

**Thread Creation:** A thread from the ForkJoinPool.commonPool() is assigned to execute the supplyAsync method block

CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> { // This code runs in a separate thread from the common pool return “hello”});

 **Non-blocking:** The main thread doesn't wait for this task(above) to complete and immediately moves on to the next line of code.

**Customization:**

* You can customize the thread pool by passing an Executor to supplyAsync() if you want to control the number of threads or use a different pool.
* Example with a custom thread pool

Executor executor = Executors.newFixedThreadPool(10);

CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> makeApiCall(), executor);

**1. Synchronous Execution:**

* **Blocking:** The method fetchUserData() is called synchronously, which blocks the main thread until the data is fetched.
* **Execution:** The processUserData() method runs only after the data has been fetched, and the main thread remains blocked during this time.

**2. Asynchronous Execution:**

* **Non-blocking:** Using CompletableFuture.supplyAsync(), the fetchUserData() method is executed asynchronously. The main thread does not wait for the result and immediately continues executing the next statement.
* **Callback:** The thenAccept() method registers a callback that will be executed when the data is fetched. This method runs in the same thread that completed the asynchronous task unless specified otherwise.
* **Continuation:** The main thread continues to execute, potentially doing other work while waiting for the asynchronous task to complete.
* **Joining:** future.join() is used to block the main thread until the asynchronous task is done, ensuring that the program doesn't exit prematurely.

**Callback Execution with thenAccept:**

By default, the thenAccept callback (or any other callback like thenApply, thenRun, etc.) will be executed by the same thread that completed the preceding stage (i.e., the thread that executed the supplyAsync task). This means it runs asynchronously but doesn't necessarily create a new thread.

**Running Callbacks in a Separate Thread:**

If you want the thenAccept (or similar callbacks) to run in a different thread, you can explicitly provide an Executor to handle the callback execution. For example:

future.thenAcceptAsync(response -> {

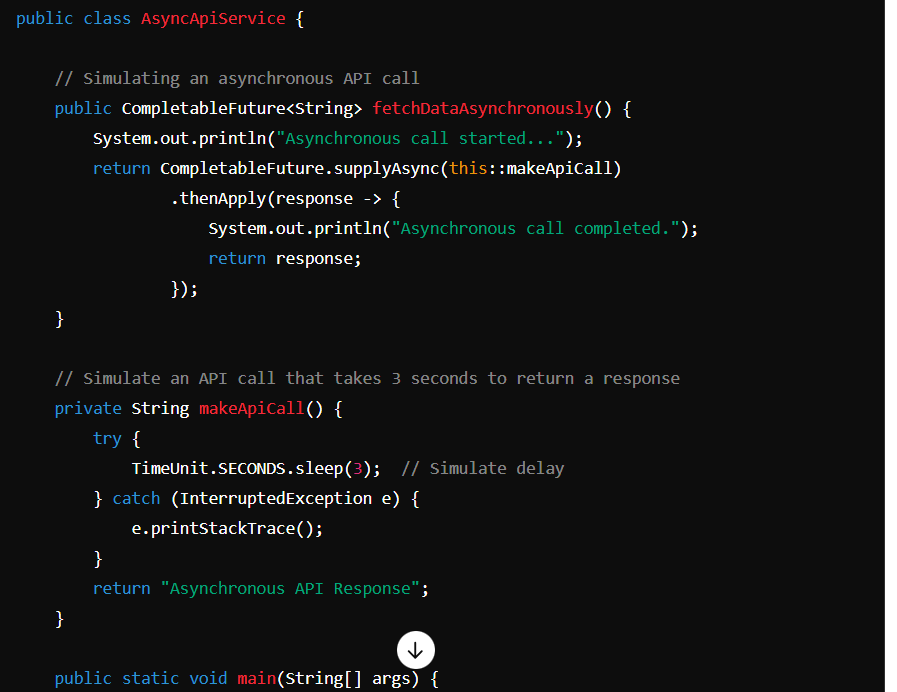
// This code runs in a separate thread from the pool provided by the executor

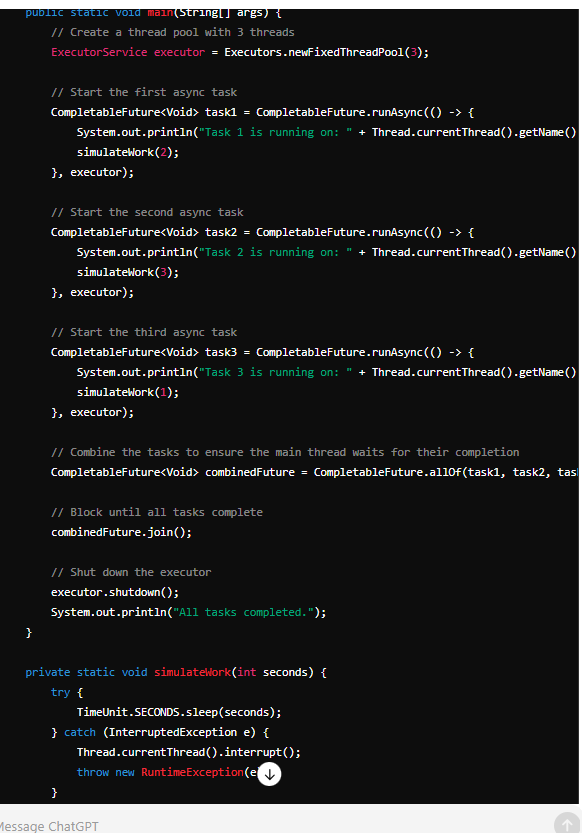
System.out.println("Response: " + response); });

* **Thread Pool:** By using thenAcceptAsync(), you're ensuring that the callback runs asynchronously, and a thread from the ForkJoinPool.commonPool() or a provided Executor will execute it.
* **Custom Executor:** Like with supplyAsync, you can also pass a custom Executor to thenAcceptAsync to control where the callback runs.

**How to Implement CompletableFuture**

Here's a simple example to illustrate how CompletableFuture can be implemented:



**Summary**

* **CompletableFuture** simplifies asynchronous programming in Java by providing a flexible, non-blocking API.
* It was introduced in **Java 8** to address the limitations of traditional concurrency tools like Future.
* **Advantages** include improved performance, easier task chaining, and better resource utilization.
* **Drawbacks** include complexity, potential debugging difficulties, and the risk of mismanaging resources.
* **Implementation** involves creating asynchronous tasks and chaining them using CompletableFuture methods like thenApply, thenAccept, and join.

This tool is particularly useful in scenarios where non-blocking, asynchronous processing is needed, such as in I/O-bound applications, microservices, and real-time data processing.

**When to Use CompletableFuture**

* **I/O-Bound Operations**: When dealing with operations that involve I/O, such as database queries, REST API calls, or file operations, where you don't want to block the main thread.
* **Parallel Processing**: When you need to perform multiple independent tasks in parallel and then combine the results.
* **Handling Long-Running Tasks**: When you have tasks that take a long time to complete, using CompletableFuture allows other tasks to proceed without waiting.
* **Microservices Communication**: For non-blocking, asynchronous communication between microservices in distributed systems.

**1. Chaining Tasks**

* Chaining allows you to run a sequence of tasks where each task starts after the previous one completes. This is achieved using methods like **thenApply, thenAccept, and thenCompose**
* thenApply: Chains a task that transforms the result of the previous task. Each step executes after the previous one completes, passing the result forward.

2. **Combining Tasks**

Combining allows you to run multiple independent tasks in parallel and combine their results once they are done. This is achieved using methods like thenCombine, thenAcceptBoth, and allOf.

* Explanation
* thenCombine: Combines the results of two independent CompletableFutures once both are complete.

**3. Exception Handling**

* Exception Handling allows you to manage errors that occur during the execution of asynchronous tasks. This is achieved using methods like exceptionally, handle, and whenComplete.

Explanation

* exceptionally: Provides a fallback mechanism when an exception occurs, allowing you to return a default value or perform some other recovery action.

**The Executor framework** in Java provides a high-level mechanism for **managing threads and allows you to decouple the task submission** from the details of how each task will be executed, including thread management, scheduling, etc. It simplifies the process of handling asynchronous tasks and enables efficient thread pool management.

**Key Components of the Executor Framework**

1. **Executor Interface**:
   * The base interface of the executor framework.
   * It has a single method execute(Runnable command), which accepts a Runnable task and executes it.

java

Copy code

public interface Executor {

void execute(Runnable command);

}

1. **ExecutorService Interface**:
   * A more sophisticated subinterface of Executor that adds lifecycle management methods and the ability to return a result from a task.
   * Provides methods like submit(), shutdown(), invokeAll(), and invokeAny().

java

Copy code

ExecutorService executorService = Executors.newFixedThreadPool(10);

1. **Executors Class**:
   * A utility class that provides factory methods for creating different types of ExecutorService instances, such as:
     + **newFixedThreadPool(int nThreads)**: Creates a thread pool with a fixed number of threads.
     + **newCachedThreadPool()**: Creates a thread pool that creates new threads as needed but will reuse previously constructed threads when available.
     + **newSingleThreadExecutor()**: Creates an executor that uses a single worker thread.
     + **newScheduledThreadPool(int corePoolSize)**: Creates a thread pool that can schedule commands to run after a given delay or to execute periodically.

java

Copy code

ExecutorService fixedThreadPool = Executors.newFixedThreadPool(5);

ExecutorService cachedThreadPool = Executors.newCachedThreadPool();

ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor();

ScheduledExecutorService scheduledExecutor = Executors.newScheduledThreadPool(5);

**Common Use Cases**

1. **Fixed Thread Pool**:
   * Best for scenarios where you have a limited number of threads and want to control the number of concurrent threads.
   * Example:

java

Copy code

ExecutorService executor = Executors.newFixedThreadPool(3);

for (int i = 0; i < 10; i++) {

executor.execute(() -> {

System.out.println(Thread.currentThread().getName() + " is executing a task.");

});

}

executor.shutdown();

1. **Cached Thread Pool**:
   * Suitable for applications with many short-lived asynchronous tasks.
   * Example:

java

Copy code

ExecutorService executor = Executors.newCachedThreadPool();

for (int i = 0; i < 10; i++) {

executor.execute(() -> {

System.out.println(Thread.currentThread().getName() + " is executing a task.");

});

}

executor.shutdown();

1. **Single Thread Executor**:
   * Ensures that tasks are executed sequentially in a single thread, useful for tasks that must be performed in order.
   * Example:

java

Copy code

ExecutorService executor = Executors.newSingleThreadExecutor();

for (int i = 0; i < 5; i++) {

executor.execute(() -> {

System.out.println(Thread.currentThread().getName() + " is executing a task.");

});

}

executor.shutdown();

1. **Scheduled Thread Pool**:
   * Allows scheduling tasks to run after a certain delay or at fixed intervals.
   * Example:

java

Copy code

ScheduledExecutorService executor = Executors.newScheduledThreadPool(2);

executor.schedule(() -> {

System.out.println("Task executed after delay");

}, 3, TimeUnit.SECONDS);

executor.scheduleAtFixedRate(() -> {

System.out.println("Task executed periodically");

}, 1, 3, TimeUnit.SECONDS);

executor.shutdown();

**Executor Lifecycle Management**

* **shutdown()**: Initiates an orderly shutdown in which previously submitted tasks are executed but no new tasks will be accepted.
* **shutdownNow()**: Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were waiting to be executed.
* **awaitTermination(long timeout, TimeUnit unit)**: Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted.

java

Copy code

executor.shutdown();

try {

if (!executor.awaitTermination(60, TimeUnit.SECONDS)) {

executor.shutdownNow();

}

} catch (InterruptedException e) {

executor.shutdownNow();

}

**Advantages of Using Executors**

* **Thread Management**: Simplifies thread management by reusing a pool of threads.
* **Task Scheduling**: Allows tasks to be scheduled with delays or to run periodically.
* **Scalability**: Improves application performance by limiting the number of active threads and efficiently managing resources.

**Summary**

The Executor framework in Java provides a powerful and flexible mechanism for managing threads, making it easier to work with multithreading. By using ExecutorService and various implementations provided by Executors, you can create, manage, and schedule tasks efficiently, making your code more maintainable and scalable.

**Permanent Generation (PermGen)**

**What is PermGen?**

* **PermGen** (Permanent Generation) **special heap space separated from the main memory The JVM keeps track of loaded class metadata in the PermGen. Additionally, the JVM stores all the static content in this memory section. This includes all the static methods, primitive variables, and references to the static objects.**
*  **Class Metadata**: Information about the classes that are loaded by the JVM. This includes the class's structure, methods, and field definitions.
*  **Method Metadata**: Details about the methods of loaded classes, including method names, signatures, and bytecode.
*  **Interned Strings**: Java automatically interns string literals to save memory. These interned strings were stored in the PermGen space.
*  **Static Variables**: All static variables in the classes were stored in the PermGen space.
*  **Classloader Data**: Information about the classloaders used to load classes.
* **Characteristics:**
* **Fixed Size**: The size of PermGen was fixed at JVM startup, and it could be adjusted using the -XX:MaxPermSize option.
* **Full GC Trigger**: A Full GC (or Major GC) would be triggered if PermGen was full, which could lead to application pauses.
* **Memory Leaks**: PermGen was prone to memory leaks, especially in applications that dynamically load and unload classes, like web applications. Classes that were no longer needed might still occupy space in PermGen if not unloaded properly.

**Common Issues:**

* **OutOfMemoryError**: If PermGen was exhausted, the JVM would throw an OutOfMemoryError: PermGen space exception.
* **Fixed Capacity**: Adjusting PermGen size required tuning, and increasing it could be cumbersome.

**Metaspace**

**What is Metaspace?**

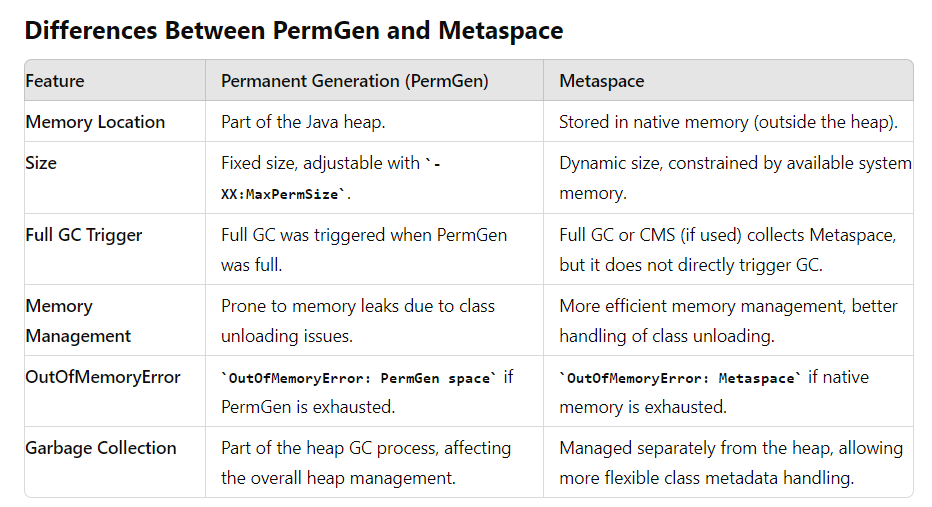
* **Metaspace** is the memory area introduced in Java 8 to replace PermGen. It is used to store class metadata, similar to PermGen, but with significant improvements.

**Characteristics:**

* **Dynamic Size**: Metaspace grows automatically as needed, constrained only by the available system memory. There is no fixed size, and it can expand dynamically.
* **Native Memory**: Metaspace is stored in native memory (outside the heap), which allows for better management of class metadata and reduces the likelihood of OutOfMemoryError related to class metadata.
* **Garbage Collection**: Metaspace is collected by the garbage collector, but it is handled differently than the heap. Class unloading in Metaspace is more efficient and does not affect heap management directly.

**Common Configurations:**

* **Max Metaspace Size**: Use the -XX:MaxMetaspaceSize option to set a maximum size for Metaspace, but this is often less critical because Metaspace does not have a fixed size like PermGen.
* **Metaspace Size Management**: You can manage the initial size of Metaspace using the -XX:MetaspaceSize option, which controls the initial amount of memory allocated for Metaspace.



**Give me some immutable classes in Java**

All Wrapper classes of Primitive data type likes Integer, Float, Double, Boolean, Character etc.. and String , List.of(), Set.of() , Map.of()

**How to create Synchronized version of collections**

Collections.synchronizedList()

Collections.synchronizedSet()

Collections.synchronizedMap()

Using **CopyOnWriteArrayList** and **CopyOnWriteArraySet**

Java provides thread-safe implementations of List and Set that are designed for cases where reads vastly outnumber writes. These are:

CopyOnWriteArrayList

CopyOnWriteArraySet

These classes make a copy of the underlying array whenever a modification is made, which allows for safe iteration without needing to manually synchronize.

Example: CopyOnWriteArrayList

**Using Concurrent Collections**

For high-concurrency environments, Java provides several concurrent collections in the java.util.concurrent package. These are specifically designed for concurrent access and include:

* ConcurrentHashMap

**Generics:**

**Generics** in Java are a powerful feature introduced in Java 5 that allows you to write more flexible and reusable code **while providing type safety at compile time.** **Generics enable you to define classes, interfaces, and methods with type parameters, which can be substituted with specific types** when the code is used. Here's an overview:

**1. Purpose of Generics**

**Type Safety**: Generics ensure that you can only use compatible types, reducing runtime errors. For example, a List<String> can only hold strings, preventing accidental insertion of other types.

Reusability: With generics, you can write a single class or method that works with different types, rather than creating separate classes or methods for each type.

**Generic Class**

**public class Box<T> {**

**private T value;**

**public void set(T value) {**

**this.value = value;**

**}**

**public T get() {**

**return value;**

**}**

**} Here, T is a type parameter that can be replaced with any class or interface type.**

**Generic Method**

public <T> void printArray(T[] array) {

for (T element : array) {

System.out.println(element);

}

}

**Bounded Type Parameters**

* You can restrict the types that can be used with generics using bounded type parameters:

public <T extends Number> void printDouble(T number) {

System.out.println(number.doubleValue());

}

Here, T can only be a subclass of Number, ensuring that the method can call doubleValue().

**5. Wildcards in Generics**

* Unbounded Wildcard: <?> represents an unknown type.

List<?> list = new ArrayList<String>();

* Bounded Wildcards:
  + ? extends T: Accepts T or any subclass of T.

public void printNumbers(List<? extends Number> list) {

for (Number num : list) {

System.out.println(num);

}

}

* + ? super T: Accepts T or any superclass of T.

public void addNumbers(List<? super Integer> list) {

list.add(10);

}

**6. Type Erasure**

* Java implements generics using a technique called type erasure. This means that generic type information is removed at runtime, and generic types are replaced with their bounds or Object if no bounds are specified. This ensures backward compatibility with older versions of Java that don't support generics.

**7. Limitations of Generics**

* No Primitive Types: Generics work with reference types, not primitive types. For example, List<int> is not allowed, but List<Integer> is.
* Cannot Instantiate Generic Types: You cannot create instances of a generic type directly (new T()) because of type erasure.
* No Static Fields with Generic Types: You cannot declare static fields of a generic type, as they are shared across all instances of the class.

**8. Real-World Example**

* A common use of generics is in the Collections Framework, where interfaces like List<E>, Map<K, V>, and classes like ArrayList<E> use generics to ensure type safety and reusability across different types.

**Summary**

Generics in Java allow you to write more flexible, reusable, and type-safe code. They are widely used in collections, algorithms, and other frameworks where the type of data varies but the operations on them remain the same.

**When to Use T**

* **Class or Interface Declaration**: Use T when defining a generic class or interface.
  + Example: public class Box<T> { ... }
* **Method Declaration**: Use T when defining a generic method.
  + Example: public <T> void addItem(T item) { ... }
* **Method Parameters**: Use T when you want the method to accept a specific type that is determined by the class or method's type parameter.
  + Example: public void setItem(T item) { ... }
* **Return Type**: Use T when you want the method to return a specific type that is determined by the class or method's type parameter.
  + Example: public T getItem() { ... }

**When to Use ?**

* **Method Parameters**: Use ? when you want the method to accept a parameter of any type, and you don't care what that type is.
  + Example: public void printList(List<?> list) { ... }
* **Return Type**: Use ? when you want the method to return a collection of any type, without specifying the exact type.
  + Example: public List<?> getList() { ... }
* **Wildcard Bounds**: Use ? extends T or ? super T when you want to specify an upper or lower bound on the type, but still allow flexibility.
  + Example: public void process(List<? extends Number> numbers) { ... }

**Summary**

* **T**: Use when defining a generic class, method, parameter, or return type where you need to specify or control the type.
* **?**: Use when you need flexibility in accepting or returning any type, especially in method parameters or return types, without specifying a particular type.

The Java Virtual Machine (JVM) is an integral part of the Java Runtime Environment (JRE) that executes Java bytecode. Understanding JVM architecture and its flow is crucial for optimizing performance and troubleshooting issues in Java applications. Here’s a detailed breakdown:

**Java Virtual Machine (JVM)**

* **Definition**: The JVM is an abstract computing machine that enables a computer to run Java programs and other languages compiled to Java bytecode.
* **Role**: It provides a runtime environment that abstracts away the underlying hardware and operating system.
* **Functionality**:
  + Executes Java bytecode, converting it into native machine code.
  + Manages memory through garbage collection.
  + Supports features like multithreading and synchronization.
* **Components**: Includes the class loader subsystem, runtime data areas (heap, stack, etc.), execution engine (interpreter and JIT compiler), and garbage collector.

**Java Runtime Environment (JRE)**

* **Definition**: The JRE is a software package that provides the libraries, Java Virtual Machine (JVM), and other components necessary to run Java applications.
* **Role**: It allows users to run Java applications but does not provide tools for developing them.
* **Components**:
  + **JVM**: The core component that executes Java bytecode.
  + **Libraries**: Includes core Java libraries (like java.lang, java.util, etc.) necessary for running Java programs.
  + **Java Plug-in**: For running Java applets in web browsers (though applets are now largely obsolete).
* **Usage**: The JRE is typically used by end-users to run Java applications and does not include development tools.

**Java Development Kit (JDK)**

* **Definition**: The JDK is a comprehensive software development kit used for developing Java applications.
* **Role**: It provides the tools and libraries needed for Java development, including the compiler, debugger, and other utilities.
* **Components**:
  + **JRE**: Included within the JDK, providing the necessary runtime environment.
  + **Java Compiler (javac)**: Converts Java source code into bytecode.
  + **Java Debugger (jdb)**: For debugging Java applications.
  + **JavaDoc**: Tool for generating API documentation from Java source code.
  + **Java Tools (javap, jar, javah, etc.)**: Additional tools for class file analysis, packaging, and header generation.
* **Usage**: The JDK is used by developers to write, compile, and debug Java applications. It includes everything needed for both development and running Java applications.

**Summary**

* **JVM**: Executes Java bytecode. It’s the runtime engine that provides an environment for Java applications.
* **JRE**: Provides the JVM and libraries needed to run Java applications. It does not include development tools.
* **JDK**: Includes the JRE and additional tools needed for Java development, such as the compiler and debugger.

To summarize, if you want to run Java applications, you need the JRE. If you want to develop Java applications, you need the JDK, which contains the JRE along with development tools.

**JVM Architecture** [**https://www.geeksforgeeks.org/jvm-works-jvm-architecture/**](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/)

1. **Class Loader Subsystem**
   * **Role**: Responsible for loading, linking, and initializing classes.

 **Loading Classes**: **The ClassLoader loads the bytecode of a class** file (.class file) into memory. It reads the file from a specified location (e.g., the file system) and converts it into an in-memory representation called a Class object.

 **Linking Classes**: After a class is loaded, it needs to be linked. Linking involves **verifying the bytecode**, **preparing the class by allocating memory for static variables**, and resolving references to other classes.

 **Initializing Classes**: **The ClassLoader initializes a class by executing its static initializers and static blocks**. This is the phase where the class is fully prepared to be used in the program.

* + **Components**:
    - **Bootstrap Class Loader**: Loads core Java libraries from the JRE. loading the core Java libraries (e.g., classes in java.lang, java.util, etc.) that are part of the Java Development Kit (JDK).
    - **Platform Class Loader**: Loads classes from the JDK’s extension directory (lib/ext).
    - **Application Class Loader**: Loads classes from the application’s classpath.

1. **Runtime Data Areas**
   * **Method Area**: This area is responsible for **holding class-level information, including static fields**, which are shared across all instances of the class. This ensures that there is only one copy of each static variable, no matter how many objects of the class are created.
   * **Heap**: Contains objects and **their instance variables**. This is where Java objects are allocated.
   * **Stack**: Each **thread has its own stack**, which stores frames. Frames **contain local variables**, operand stacks, and dynamic links.
   * **Program Counter (PC) Register**: Store address of current execution instruction of a thread. Obviously, each thread has separate PC Registers
   * **Native Method Stack**: Supports native methods (methods written in languages like C or C++).
2. **Execution Engine**
   * **Interpreter**: Reads and executes bytecode instructions one at a time. It’s straightforward but can be slow.
   * **Just-In-Time (JIT) Compiler**: Converts bytecode into native machine code at runtime to improve performance. The compiled code is cached for subsequent executions.
3. **Native Interface**
   * **Java Native Interface (JNI)**: Provides a way for Java code running in the JVM to interact with native applications and libraries written in other languages like C or C++.
4. **Garbage Collector**
   * **Role**: Automatically manages memory by reclaiming memory occupied by objects that are no longer in use.
   * **Types**:
     + **Serial Garbage Collector**: Uses a single thread for garbage collection. Serial Garbage collector is well-matched for **single-threaded environments**. It uses the **only** thread for garbage collection. It works by holding all the threads of an application. It means that threads of the application **freeze** by the serial garbage collector during the garbage collection process and the process is known as **stop the world event**. Avoid the use of serial GC in the server environment. We can use it for simple programs. If you want to use the serial garbage collector, execute the **-XX:+UseSerialGC** JVM argument to activate it.
     + **Parallel Garbage Collector**: **Uses multiple threads for garbage collection**. Parallel Garbage Collector is the **default GC** used by the JVM. The working of the parallel garbage collector is the same as the serial garbage collector. The only difference between serial and parallel garbage collector is that serial garbage collector uses a **single** thread for garbage collection process while the parallel garbage collector uses **multiple** threads for the garbage collection. Parallel GC can use multiple CPUs to speed up the application throughput. So, it is also known as **throughput collector**. It is used if we want to execute a long process (like batch processing) and where long pauses are acceptable. If you want to use the parallel garbage collector, execute the **-XX:+UseParallelGC** JVM argument to activate it.
     + **Concurrent Mark-Sweep (CMS) Garbage Collector**: Minimizes pauses by performing most of its work concurrently with the application.
     + **G1 (Garbage-First) Garbage Collector**: Divides the heap into regions and prioritizes the collection of regions with the most garbage.

**JVM Flow**

1. **Loading Phase**
   * The **Class Loader Subsystem** loads the required class files into memory.
   * **Class Loading**: The class loader loads the class files from the filesystem or network.
   * **Linking**: The class is linked, which involves verification, preparation (allocating memory for class variables), and resolution (resolving symbolic references).
2. **Initialization Phase**
   * **Initialization**: The JVM initializes the class. This involves setting the default values for class variables and executing static initializers.
3. **Execution Phase**
   * **Bytecode Interpretation**: The JVM reads and executes bytecode instructions using the interpreter or JIT compiler.
   * **Method Invocation**: Methods are invoked, and the corresponding bytecode is executed.
   * **Runtime Data Area Management**: The JVM uses the runtime data areas (heap, stack, etc.) to manage and access the data needed for execution.
4. **Garbage Collection Phase**
   * The **Garbage Collector** periodically runs to clean up memory by removing objects that are no longer referenced.
5. **Shutdown Phase**
   * **Termination**: When the JVM is shutting down, it performs finalization tasks such as executing any registered shutdown hooks and releasing resources.

**Summary**

* The **Class Loader Subsystem** handles class loading and initialization.
* The **Runtime Data Areas** manage the data required for execution.
* The **Execution Engine** interprets or compiles bytecode into native instructions.
* The **Native Interface** allows interaction with native code.
* The **Garbage Collector** manages memory automatically.
* The **JVM Flow** involves loading, initializing, executing, and finally shutting down.

This architecture ensures that Java applications are portable and can run on any device that has a JVM implementation.

4o mini

**You said:**

the above is for java7 or 8

**ChatGPT said:**

ChatGPT

The explanation I provided is applicable to both Java 7 and Java 8, as the fundamental architecture of the JVM has remained largely consistent across these versions. However, there are some enhancements and additional features introduced in Java 8 that affect the JVM architecture and performance. Here’s a summary of Java 8-specific changes:

**Java 8 Enhancements**

1. **Class Loader Subsystem**
   * **No significant changes** in the class loader subsystem between Java 7 and Java 8.
2. **Runtime Data Areas**
   * **Metaspace**: Java 8 replaced the PermGen (Permanent Generation) space with Metaspace. Unlike PermGen, Metaspace is not part of the heap and is allocated in native memory. This change helps avoid OutOfMemoryError related to PermGen space and provides better management of class metadata.
3. **Execution Engine**
   * **Lambda Expressions and Method References**: Java 8 introduced lambda expressions and method references, which require changes in the execution engine to handle these new constructs. The JVM supports these features with new bytecode instructions.
4. **Native Interface**
   * **No significant changes** in the Java Native Interface (JNI) between Java 7 and Java 8.
5. **Garbage Collector**
   * **G1 Garbage Collector Enhancements**: Java 8 improved the G1 garbage collector introduced in Java 7, making it more efficient in managing heap space and reducing garbage collection pauses.
6. **New APIs and Libraries**
   * **Streams API**: Java 8 introduced the Streams API, which is used for processing sequences of elements (like collections) in a functional style. This addition affects how bytecode is generated and executed.
   * **Default Methods**: Interfaces can now have default methods with implementations, affecting class loading and bytecode processing.
7. **JVM Changes**
   * **Java 8 also introduced new JVM options** related to Metaspace and garbage collection. For example, options for configuring Metaspace (-XX:MaxMetaspaceSize) and improved GC logging options (-Xloggc).

**Summary**

While the core architecture of the JVM remains similar between Java 7 and Java 8, Java 8 introduced significant enhancements, particularly in memory management (Metaspace), garbage collection (G1 improvements), and language features (lambda expressions and Streams API). These changes are aimed at improving performance, memory management, and the developer experience.

Java 11, released in September 2018, brought several new features and enhancements. Here's a summary of the key features introduced in Java 11:

**1. New String Methods**

* **isBlank()**: Checks if a string is empty or contains only white space.
* **lines()**: Returns a stream of lines extracted from a string, separated by line terminators.
* **strip(), stripLeading(), stripTrailing()**: Removes white spaces from the beginning and end of a string.
* **repeat(int count)**: Repeats the string a specified number of times.

**2. Local-Variable Syntax for Lambda Parameters**

* Allows you to use var to declare the type of lambda parameters, which can be useful for annotations or for more concise code.

java

Copy code

(var s1, var s2) -> s1 + s2;

**3. HTTP Client (Standard)**

* The new HttpClient API, introduced as an incubator module in Java 9, was standardized in Java 11. It provides a modern and feature-rich API to perform HTTP requests.

java

Copy code

HttpClient client = HttpClient.newHttpClient();

HttpRequest request = HttpRequest.newBuilder()

.uri(URI.create("https://example.com"))

.build();

HttpResponse<String> response = client.send(request, BodyHandlers.ofString());

**4. Removal of Java EE and CORBA Modules**

* Java 11 removed the deprecated Java EE (e.g., javax.xml.bind) and CORBA modules that were part of the standard library.

**5. Nest-Based Access Control**

* This feature introduces a new access control context to support private access in nested classes, improving encapsulation and maintaining backward compatibility.

**6. Running Java Files with java Command**

* You can run a single-file Java program without compiling it first using the java command.

bash

Copy code

java HelloWorld.java

**7. Pattern Matching with Pattern.compile()**

* Enhancements to the Pattern class allow for more efficient pattern matching, especially in scenarios where the same pattern is used repeatedly.

**8. Garbage Collection (GC) Improvements**

* Java 11 introduced the **Epsilon GC**, a no-op garbage collector that is useful for performance testing. It also included enhancements to the existing G1 garbage collector.

**9. Flight Recorder**

* Java Flight Recorder (JFR), previously a commercial feature in Oracle JDK, was open-sourced and included in Java 11. It’s a low-overhead data collection framework for profiling and monitoring Java applications.

**10. New Files Methods**

* New methods like writeString(Path, String), readString(Path) were added to the Files class, making file handling more straightforward.

**11. Deprecations and Removals**

* Various older features and APIs were deprecated or removed, such as the Java Applet API, which was deprecated for removal.

These are some of the key features and enhancements that Java 11 introduced. It was also the first long-term support (LTS) release after Java 8, making it significant for production use.

**12.Epsilon GC (JEP 318)**

* **Introduction**: Epsilon GC was introduced as a no-op garbage collector in Java 11.
* **Purpose**: Epsilon GC is designed for situations where no garbage collection is needed, such as performance testing, benchmarking, or when managing memory manually.

**Java 17:**

**Sealed classes** were introduced in Java 15 as a preview feature and became a standard feature in Java 17. Sealed classes and interfaces allow you to restrict which other classes or interfaces can extend or implement them. This is useful for defining a more controlled and predictable inheritance hierarchy.

**What are Sealed Classes?**

A **sealed class** is a class that restricts which other classes can extend it. When you declare a class as sealed, you must explicitly specify the classes that are allowed to subclass it using the permits clause. This restriction helps in maintaining a controlled class hierarchy and can be used to enforce business logic or design patterns more strictly.

**Why Use Sealed Classes?**

* **Control over Inheritance**: Sealed classes give you the power to limit inheritance. This is particularly useful in domain modeling, where you want to define a closed set of types.
* **Exhaustiveness in Switch Statements**: When used with enums or pattern matching (like switch expressions), sealed classes allow the compiler to ensure that all possible cases are covered, reducing the likelihood of runtime errors.
* **Better Code Maintenance**: By controlling which classes can extend a sealed class, you reduce the risk of unexpected extensions, making the code easier to maintain.

**How to Declare a Sealed Class**

To declare a sealed class, use the sealed modifier in the class declaration and specify the allowed subclasses with the permits keyword.

**Example**

java

Copy code

public **sealed class** Shape

**permits** Circle, Rectangle, Square {

// common methods and properties for all shapes

}

public final class Circle extends Shape {

// Circle-specific implementation

}

public final class Rectangle extends Shape {

// Rectangle-specific implementation

}

public non-sealed class Square extends Shape {

// Square-specific implementation

}

**Components of Sealed Classes**

1. **sealed Modifier**:
   * The class is marked with the sealed keyword.
   * Only the classes specified in the permits clause can extend this class.
2. **permits Clause**:
   * This specifies the classes that are allowed to extend the sealed class. In the example above, only Circle, Rectangle, and Square can extend Shape.
3. **Subclasses of Sealed Classes**:
   * **Final Subclasses**: A subclass of a sealed class can be declared final, meaning it cannot be further extended. For example, Circle and Rectangle are final, so no other class can extend them.
   * **Non-sealed Subclasses**: A subclass can be declared non-sealed, which allows it to be extended by other classes. For example, Square is non-sealed, so other classes can extend it.
   * **Sealed Subclasses**: A subclass can also be sealed, continuing the inheritance restriction. This allows further control over the class hierarchy.

**Sealed Interfaces**

Sealed classes are not limited to classes; you can also seal interfaces:

java

Copy code

public sealed interface Vehicle

permits Car, Truck, Bike {

// methods common to all vehicles

}

public final class Car implements Vehicle {

// Car-specific implementation

}

public final class Truck implements Vehicle {

// Truck-specific implementation

}

public non-sealed class Bike implements Vehicle {

// Bike-specific implementation

}

**Benefits of Using Sealed Classes**

1. **Exhaustiveness**: When used with pattern matching, sealed classes allow the compiler to check that all possible subclasses are handled, reducing the risk of missing a case.

java

Copy code

public String describeShape(Shape shape) {

return switch (shape) {

case Circle c -> "This is a circle";

case Rectangle r -> "This is a rectangle";

case Square s -> "This is a square";

};

}

* + If a new subclass of Shape is added, the compiler will flag an error in the switch statement, ensuring all cases are considered.

1. **Control**: You control the inheritance, ensuring that only the intended classes can extend your sealed class or implement your sealed interface.
2. **Design Clarity**: Sealed classes make it clear which classes are intended to be part of a particular hierarchy, leading to better design and understanding of the code.

**Use Cases**

* **Domain Modeling**: When modeling a domain where a specific set of related types is known and fixed, sealed classes help encapsulate this hierarchy.
* **API Design**: When designing APIs, you can use sealed classes to prevent unauthorized extensions, ensuring that your API behaves as expected.
* **Pattern Matching**: Sealed classes work well with Java's evolving pattern matching features, ensuring that all cases are handled.

**Conclusion**

Sealed classes in Java provide a powerful way to control inheritance and class hierarchies. They allow developers to create more predictable and maintainable code by explicitly specifying which classes can extend or implement a particular class or interface. This feature is particularly useful in domain modeling, API design, and ensuring completeness in pattern matching.

1. **Pattern Matching for switch (Preview)**

* Extends pattern matching to switch statements and expressions, allowing for more concise and expressive code. It enables the use of pattern matching logic directly within switch cases.

// in simple way Parent in switch will allow call the child class based on condition it will switch

Copy code

switch (object) {

case String s -> System.out.println("It's a string: " + s);

case Integer i -> System.out.println("It's an integer: " + i);

default -> System.out.println("Unknown type");

}

**3.Removal of the Applet API**

* The Applet API, long deprecated, was finally removed in Java 17. This is part of the ongoing effort to modernize the JDK by eliminating outdated APIs

**4.Enhanced Pseudo-Random Number Generators**

The enhancements include the introduction of new interfaces and classes that provide a more extensible and flexible framework for pseudo-random number generation with new algorithms.

**5.Z Garbage Collector (ZGC) Improvements**

* Java 17 further improves the Z Garbage Collector, making it more efficient and ready for wider adoption in production environments.

1. **Vector API (Java 17 and later)**
2. **Purpose**:
   * The Vector API is designed for **high-performance numeric computations** by leveraging vectorized hardware instructions. It aims to optimize operations on multiple data elements simultaneously (e.g., SIMD - Single Instruction, Multiple Data).
3. **Usage**:
   * It is used for **low-level vector operations** and numerical processing, where you need to perform the same operation on multiple data points in parallel.
4. **Implementation**:
   * The Vector API provides classes like Vector128 for handling vector operations at the hardware level. It abstracts hardware details to make it easier to use hardware vector instructions in Java code.

**ZGC (Z Garbage Collector)** is a garbage collection algorithm introduced in Java 11 as an experimental feature and made production-ready in later versions. It is designed to handle large heaps with minimal pause times, making it suitable for applications requiring low-latency and high-throughput.

**Key Features of ZGC**

1. **Low Latency**:
   * ZGC aims for very low pause times, typically in the range of a few milliseconds, regardless of heap size. This makes it ideal for applications where even short GC pauses can impact performance or user experience.
2. **Concurrent and Incremental**:
   * ZGC performs most of its work concurrently with the application threads, minimizing pauses. It uses incremental compaction, breaking down the compaction process into smaller steps to reduce pause times.
3. **Region-based Memory Management**:
   * It divides the heap into regions, which helps in managing memory more efficiently. This region-based approach allows ZGC to perform operations on individual regions rather than the entire heap.
4. **Garbage Collection Phases**:
   * **Relocation**: Moves live objects to new memory locations, which helps in compacting the heap.
   * **Remapping**: Updates references to the new memory locations.
   * **Collection**: Frees up memory from regions that are no longer used.
5. **Concurrent Marking**:
   * ZGC uses a concurrent marking phase that runs in parallel with application threads. This phase identifies live objects and marks them for further processing.
6. **Region-Based Compaction**:
   * Unlike traditional GC algorithms that may compact the entire heap, ZGC compacts only the regions that require it. This reduces the overhead and impact on pause times.
7. **Use of Safepoints**:
   * ZGC uses safepoints to ensure that all threads are paused briefly when necessary. These safepoints are typically very short and used to ensure consistency during GC operations.

**How ZGC Works**

1. **Initial Marking**:
   * ZGC starts by marking live objects. This phase is done concurrently with application threads.
2. **Relocation**:
   * Live objects are relocated to new regions in memory. This step helps in reducing fragmentation and making free space available for new objects.
3. **Remapping**:
   * References to objects are updated to point to their new locations in memory.
4. **Finalization**:
   * ZGC performs a final cleanup of unused memory regions and ensures that all references are correctly updated.

**Configuration**

You can enable and configure ZGC using JVM flags. Some common flags include:

* -XX:+UseZGC: Enables ZGC.
* -Xmx<size>: Sets the maximum heap size.
* -XX:ConcGCThreads=<number>: Specifies the number of threads used for concurrent phases.
* -XX:ZAllocationSpikeTolerance=<value>: Adjusts the tolerance for memory allocation spikes.

**Example**

To run a Java application with ZGC, you can use the following command:

bash

Copy code

java -XX:+UseZGC -Xmx4G -jar your-application.jar

This command configures the JVM to use ZGC with a maximum heap size of 4 GB.

**Advantages**

1. **Low Pause Times**: Provides consistent and low pause times, suitable for latency-sensitive applications.
2. **Scalability**: Handles large heaps efficiently, scaling well with application requirements.
3. **Concurrent Operation**: Performs most GC operations concurrently, minimizing impact on application performance.

**Limitations**

1. **Experimental Features**: ZGC was considered experimental in earlier Java versions, but it has been made production-ready in later versions.
2. **Resource Consumption**: May use more resources compared to traditional GC algorithms, depending on the workload and heap size.
3. **Compatibility**: As with any new technology, there may be compatibility considerations with certain applications or libraries.

**Conclusion**

ZGC is a modern garbage collector designed for applications requiring low-latency and large-heap management. By performing most operations concurrently and minimizing pause times, it provides a compelling option for applications where traditional GC algorithms might introduce unacceptable delays. Its region-based and incremental approach helps manage memory efficiently and ensures minimal impact on application performance.

[**Garbage collection**](https://www.javatpoint.com/Garbage-Collection) is a mechanism that provides automatic memory management. It is done by the JVM. It need not to handle object allocation and deallocation by the programmer. In the previous sections, we have also discussed how garbage collection works.

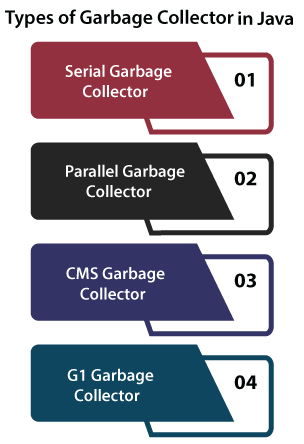
If you are not introduced to garbage collection, we suggest that first, go through the garbage collection, [**Java heap**](https://www.javatpoint.com/java-heap), and [**memory management**](https://www.javatpoint.com/memory-management-in-java). In this section, we will discuss the **types of garbage collection in Java**.

Types of Garbage Collector

There are **four** types of the garbage collector in [Java](https://www.javatpoint.com/java-tutorial) that can be used according to the requirement:

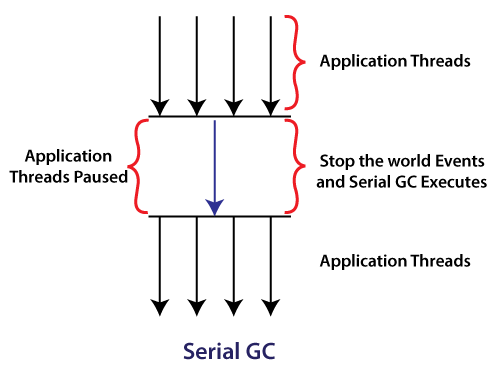
* Serial Garbage Collector
* Parallel Garbage Collector
* Concurrent Mark Sweep (CMS) Garbage Collector
* Garbage First (G1) Garbage Collector

The performance and working of each garbage collector are completely different. It has its own pros and cons. Java allows us to choose any one garbage collector that is to be used by the [JVM](https://www.javatpoint.com/jvm-java-virtual-machine). For the selection of GC, we need to pass JVM arguments. It is difficult to select the right garbage collector for the application. Let's discuss each garbage collector one by one.



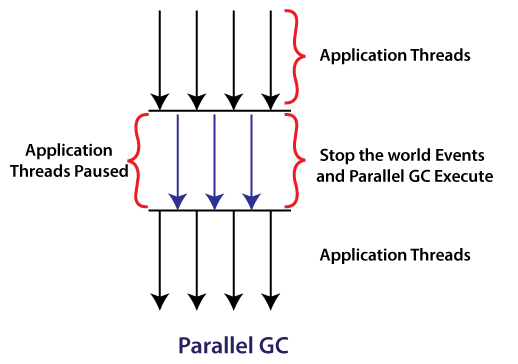
Serial Garbage Collector

Serial Garbage collector is well-matched for **single-threaded environments**. It uses the **only** thread for garbage collection. It works by holding all the threads of an application. It means that threads of the application **freeze** by the serial garbage collector during the garbage collection process and the process is known as **stop the world event**. Avoid the use of serial GC in the server environment. We can use it for simple programs. If you want to use the serial garbage collector, execute the **-XX:+UseSerialGC** JVM argument to activate it.



Parallel Garbage Collector

Parallel Garbage Collector is the **default GC** used by the JVM. The working of the parallel garbage collector is the same as the serial garbage collector. The only difference between serial and parallel garbage collector is that serial garbage collector uses a **single** thread for garbage collection process while the parallel garbage collector uses **multiple** threads for the garbage collection. Parallel GC can use multiple CPUs to speed up the application throughput. So, it is also known as **throughput collector**. It is used if we want to execute a long process (like batch processing) and where long pauses are acceptable. If you want to use the parallel garbage collector, execute the **-XX:+UseParallelGC** JVM argument to activate it.



We can also use the following JVM arguments in parallel GC:

|  |  |
| --- | --- |
| **JVM Argument** | **Description** |
| -XX:ParallelGCThreads=<n> | It controls the number of GC threads (n). |
| -XX:MaxGCPauseMillis=<t> | It specifies the maximum pause time\*. |
| -XX:GCTimeRatio=<n> | It specifies the maximum throughput target\*\*. |

**\*Pause Time:**The gap between two GC.

**\*\*Throughput Target:** The time spent during the garbage collection versus the time spent outside of garbage collection is called throughput target.

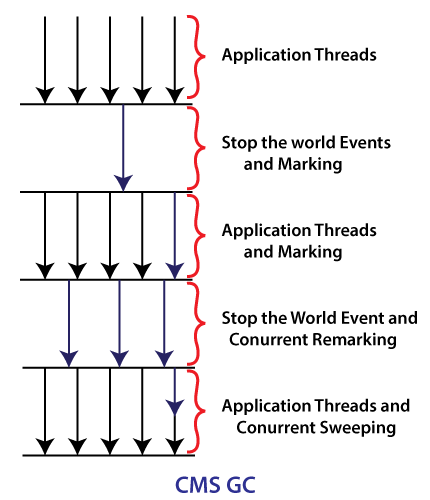
Concurrent Mark and Sweep (CMS) Garbage Collector

CMS uses multiple threads that scan the heap and during the scanning, it marks the instances for eviction, after scanning, it sweeps the marked instances. It does not freeze the application's threads during the garbage collection. GC threads concurrently execute with the application's threads. For this reason, it uses more CPU in comparison to other GC. It is also known as the **concurrent low pause collector**. It also freezes all the threads of the application only if it satisfies the following two scenarios:

* while marking the referenced objects in the tenured generation region.
* if any change is made to the heap memory in parallel during the garbage collection process.

AD

We can use multiple CPUs for better application throughput. We should use a CMS garbage collector if we have more CPUs for use. Hence, it has an advantage over the parallel garbage collector. If you want to use a CMS garbage collector, execute the **-XX:+USeParNewGC**JVM argument to activate it. We can also set the number of GC threads by using the **-XX:ParallelCMSThreads=<n>** JVM argument.



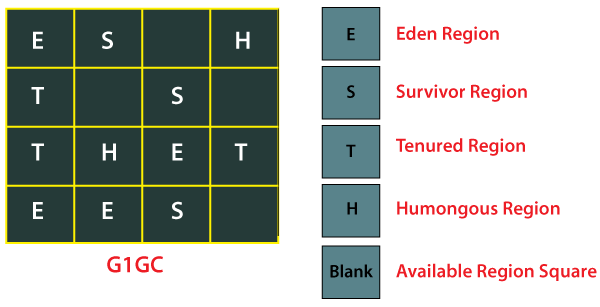
Note: The JVM argument -XX:+UseConcMarkSweepGC has been deprecated because a warning message is issued when it is requested on the command line.

Garbage First (G1) Garbage Collector

The G1 garbage collector is used if we have a large (more than 4GB) memory (heap space). It divides the heap into equal-sized (usually 1MB to 32MB) chunks, prioritizes them, and then performs the parallel garbage collection on that chunks based on the priority.

The Eden, survivors, and old areas use this equal-sized region for the memory allocation of the objects. Apart from these memory regions, there are two more types of regions presented in the G1 GC:

* **Humongous:** It is used if the object sized is large.
* **Available:** It represents the unoccupied space.



G1 GC shows a concurrent global marking phase to determine the live and dead objects throughout the heap. After the completion of the mark phase, G1 collects the information of regions that contains the most garbage objects. After that these regions are swept first. If you want to use the G1 garbage collector, execute the **-XX:+UseG1GC** JVM argument to activate it.

Stop the World Event

It is a situation when the garbage collector performs the garbage collection (GC) and stops all the application's threads until the GC process is not completed. The process is known as **Stop the World (STW)**events.

AD

Improvement in Garbage Collector Since Java 8

In Java 8, the G1 garbage collector has been updated. The updated GC provides the **-XX:+UseStringDeduplication** JVM argument that optimizes the heap memory. It removes the duplicate String values to a single char[] array.

Garbage Collection JVM Options

Java garbage collection key options are as follows:

AD

JVM Arguments

The table describes the arguments that can be used to instruct the JVM.

|  |  |
| --- | --- |
| **Option** | **Description** |
| -XX:+UseSerialGC | Serial Garbage Collector |
| -XX:+UseParallelGC | Parallel Garbage Collector |
| -XX:+UseConcMarkSweepGC | CMS Garbage Collector |
| -XX:ParallelCMSThreads= | CMS Collector - number of threads to use |
| -XX:+UseG1GC | G1 Garbage Collector |
| **Some Other Important Arguments** | |
| -XX:InitiatingHeapOccupancyPercent=<n> | It controls the **heap occupancy** after starting the concurrent cycle. Here, n is the percentage of heap space. The default percentage is **45**. |
| -XX:G1MixedGCLiveThresholdPercent=<t> | If a live object in the OLD region exists with a value greater than or equal to this option, it is excluded from the GC object. Here, t is the time in **milliseconds**. The default value is **65**. |
| -XX:G1HeapWastePercent=<r> | It specifies how many **regions** are allowed to be wasted. Here, r is the number of regions. The default value is **10**. |

GC Optimization Options

|  |  |
| --- | --- |
| **Option** | **Description** |
| -Xms | It specifies the initial heap size. |
| -Xmx | It maximizes the heap size. |
| -Xmn | It is used to specify the size of the Young Generation. |
| -XX:PermSize | Initial Permanent Generation size |
| -XX:MaxPermSize | It is used to specify the maximum size of Permanent Generation (PermGen). |

Usage of JVM GC Options

1. java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseSerialGC -jar java-application.jar

We have discussed all four types of Java garbage collectors. But it is difficult to select one because it depends on the application scenario, availability of hardware, and throughput requirements.

Ref: https://www.javatpoint.com/types-of-garbage-collector-in-java