### **🔹 1. What is Multithreading in Java?**

* **Multithreading** is the process of executing multiple threads concurrently within a single program (process).
* It helps achieve **parallelism** (multiple tasks running seemingly at the same time) and **better CPU utilization**.

### **🔹 2. What are Threads?**

* A **thread** is the smallest unit of execution in a process.
* Each thread has its **own program counter, registers, and stack**, but shares the **heap memory** of the process with other threads.
* In Java, Thread class and Runnable interface are used to create threads.

### **🔹 3. Difference Between Multithreading and Multiprocessing**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Multithreading** | **Multiprocessing** |
| Definition | Multiple threads in one process | Multiple processes running independently |
| Memory | Threads share memory (heap) | Processes have separate memory space |
| Context Switching | Faster (less overhead) | Slower (more overhead) |
| Communication | Easy (shared memory) | Hard (IPC required) |
| Example | Java threads | Running multiple JVM instances |

### **🔹 4. How to Make a Thread-Safe Queue?**

* Use **Concurrent Collections** from java.util.concurrent:

Queue<Integer> queue = new ConcurrentLinkedQueue<>();  
BlockingQueue<Integer> blockingQueue = new ArrayBlockingQueue<>(100);

* Or wrap with synchronization:

Queue<Integer> safeQueue = Collections.synchronizedList(new LinkedList<>());

### **🔹 5. How Can We Make a Piece of Code Thread Safe?**

* **Synchronization**: Use synchronized blocks/methods.
* **Locks**: Use ReentrantLock, ReadWriteLock.
* **Atomic classes**: Use AtomicInteger, AtomicReference.
* **Immutable objects**: Objects whose state cannot change are inherently thread safe.
* **Volatile keyword**: For visibility of changes across threads.

### **🔹 6. What is a Daemon Thread?**

* **Daemon thread** is a low-priority background thread that runs for supporting tasks (like GC).
* JVM exits when only daemon threads remain.
* Example:

Thread t = new Thread(() -> {  
 while(true) System.out.println("Daemon running...");  
});  
t.setDaemon(true);  
t.start();

### **🔹 7. Does Each Thread Have Its Own Stack?**

✅ **Yes**.

* Each thread has its own **call stack** (local variables, method calls).
* But all threads share the **heap memory** of the process.

### **🔹 8. Available Thread States (from**

### **Thread.State enum)**

1. **NEW** → Thread created, not started.
2. **RUNNABLE** → Thread ready to run, waiting for CPU.
3. **BLOCKED** → Waiting to acquire a monitor lock.
4. **WAITING** → Waiting indefinitely for another thread’s signal.
5. **TIMED\_WAITING** → Waiting with a timeout.
6. **TERMINATED** → Thread has finished execution.

### **🔹 9. What is Thread Group? Why Not Use It?**

* **ThreadGroup**: A way to group multiple threads together for management.
* Example: ThreadGroup group = new ThreadGroup("MyGroup");
* **Why not use it?**
  + It’s considered **obsolete** (introduced in early Java for security).
  + Modern APIs (ExecutorService, ForkJoinPool) are preferred for better control and flexibility.

### **🔹 10. What is Context-Switching in Multi-threading?**

* **Context switch** is when CPU saves the state of one thread and loads another thread’s state to resume execution.
* Involves saving program counter, registers, stack.
* It causes **overhead**, but allows multiple threads to share CPU time.

### **🔹 11. How Code Should Be Made Thread Safe?**

* Use **concurrent utilities** (ConcurrentHashMap, CopyOnWriteArrayList).
* Use **synchronization** only when necessary (avoid bottlenecks).
* Use **atomic operations** instead of locks when possible.
* Prefer **immutable objects**.
* Use **Executor framework** to manage threads instead of manual creation.

### **🔹 12. What is Concurrency?**

* **Concurrency** is the ability to handle multiple tasks at once (not necessarily simultaneously).
* In Java, concurrency is supported through **multithreading**, thread pools, and asynchronous APIs.

### **🔹 13. Difference Between Threads and Processes**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Threads** | **Processes** |
| Definition | Smallest execution unit inside a process | Independent execution unit |
| Memory | Share heap, own stack | Separate memory space |
| Communication | Easy (shared memory) | Hard (IPC needed) |
| Creation | Lightweight | Heavyweight |
| Example | Multiple threads in a Java app | Running Chrome + IntelliJ |

Memory Allocation

## **🔹 Memory Model in a Multithreaded Java Program**

When you start a **Java process (JVM instance)**, the **OS allocates memory** for it. Inside that process, threads share some areas and have their own private areas.

### **1. Shared Memory (All Threads Share)**

* **Heap**
  + Stores objects created via new.
  + Shared by all threads.
  + Example: Two threads can access the same ArrayList in heap.
* **Method Area (a.k.a. MetaSpace in Java 8+)**
  + Stores class structures (fields, methods, static variables).
  + Shared across all threads.

### **2. Thread-Private Memory (Each Thread Has Its Own)**

* **Program Counter (PC) Register**
  + Each thread has a small register storing the address of the current JVM instruction.
* **Java Stack**
  + Each thread has its own stack.
  + Stores **method frames**: local variables, partial results, and return addresses.
  + Example:

void foo(int x) {  
 int y = 10; // y stored in thread’s stack  
}

* + If 2 threads call foo(), each has its own copy of x and y.
* **Native Stack (C Stack)**
  + Used when Java calls native methods via JNI.
  + Each thread has its own.

### **3.**

### **Illustration**

PROCESS (JVM)  
 ├── Shared Memory  
 │ ├── Heap (Objects, shared data)  
 │ └── Method Area (Class metadata, static variables)  
 │  
 ├── Thread 1  
 │ ├── PC Register  
 │ ├── Java Stack (local vars, method calls)  
 │ └── Native Stack  
 │  
 ├── Thread 2  
 │ ├── PC Register  
 │ ├── Java Stack  
 │ └── Native Stack  
 │  
 └── Thread N ...

### **4. Implications**

* ✅ Shared memory (heap) → Risk of **race conditions**, so synchronization is needed.
* ✅ Private stacks → No need for synchronization for **local variables**.
* ✅ Objects in heap → Accessible by all threads if reference is shared.

Great question 👍 — **Runnable** and **Callable** are two core Java interfaces used for representing tasks that run in separate threads. Let’s break it down clearly:

## **🔹 1. Runnable**

* **Package**: java.lang
* **Method**:

void run();

* **Return Type**: void (no return value).
* **Exception Handling**: Cannot throw checked exceptions.
* **Usage**: Suitable when the task **does not return a result**.

✅ Example:

Runnable task = () -> {  
 System.out.println("Task is running...");  
};  
  
Thread t = new Thread(task);  
t.start();

## **🔹 2. Callable<V>**

* **Package**: java.util.concurrent
* **Method**:V call() throws Exception;
* **Return Type**: V (a result).
* **Exception Handling**: Can throw checked exceptions.
* **Usage**: Suitable when the task **returns a result or may throw exceptions**.

✅ Example:

import java.util.concurrent.\*;  
  
Callable<Integer> task = () -> {  
 Thread.sleep(1000);  
 return 42; // returns a result  
};  
  
ExecutorService executor = Executors.newSingleThreadExecutor();  
Future<Integer> future = executor.submit(task);  
  
System.out.println("Result: " + future.get()); // waits and prints 42  
executor.shutdown();

## **🔹 3. Key Differences**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Runnable** | **Callable** |
| Method | run() | call() |
| Return Value | No (void) | Yes (generic type V) |
| Exceptions | Cannot throw checked | Can throw checked |
| Package | java.lang | java.util.concurrent |
| Use Case | Fire-and-forget tasks | Tasks that produce results |

## **🔹 4. Relationship**

* You can submit both Runnable and Callable tasks to an ExecutorService.
* For Runnable, submit() returns a Future<?> with null result.
* For Callable, submit() returns a Future<V> with actual result.