### **Multitasking in Java**

Multitasking in Java means running multiple tasks or processes at the same time. It is achieved using multithreading, where a program is divided into smaller threads that run independently, or multiprocessing, where multiple processes run simultaneously to improve efficiency.

**Types of Multitasking in Java**

Java supports multitasking in two primary forms:

* **Processed Based**
* **Task Based**

Let’s begin with **Process-Based:** A **process** is a program that is currently being executed by the operating system. It includes the program code (often referred to as the text section) and its current activity, which is represented by the value of the program counter, registers, and variables.

### 

**Real World Analogy: Theater**

Think of a process like a theater, where each play represents a process. Within that theater, there can be several actors (threads) performing different roles in the play. However, at any given moment, only one actor is on stage, delivering their lines while the others wait for their turn. This shows how, even though there are multiple threads within a single process, only one thread is active at a time.

Real Time Examples:

* **Web Browser**: When you open a web browser, each tab you open can be considered a separate process. Each tab runs independently, allowing you to browse different websites simultaneously. For instance, if one tab crashes or becomes unresponsive, the other tabs may continue to work.
* **Text Editor**: When you use a text editor like Microsoft Word, the program itself is a process. If you open multiple documents, each document can run in its own process, allowing you to edit different files at the same time without affecting the others.
* **Media Player**: When you play a video using a media player, the player runs as a process. If you decide to stream music in another application while watching the video, both applications run as separate processes, allowing you to multitask.

### 

**Task-Based /Thread-Based**: A **task** is a smaller unit within a process, often referred to as a thread. Multiple threads within the same process share the same memory space, making them lightweight and faster to switch between.

**Real World Analogy: Project Manager**

Think of a project manager overseeing multiple projects. Each project represents a task. The manager can assign different team members to various projects at the same time, ensuring that all tasks progress concurrently.

**Example**: **In a software context** :

A process is like a full program, such as opening a web browser, while a task (thread) could be the browser loading different tabs simultaneously.

Consider a photo editing application that allows you to apply filters, crop images, and adjust brightness simultaneously. Each of these actions is a task that can run independently while you continue to work on your project.

**Process-Based vs. Task-Based Multitasking**

| Processes are independent applications running in separate memory spaces. | Threads are lightweight sub-tasks within a single process, sharing the same memory space. |
| --- | --- |

**Example**: Opening multiple programs (e.g., a browser, word processor) on a computer is process-based multitasking, while loading multiple tabs within a browser is thread-based multitasking.

### **What is Concurrency?**

Concurrency in Java allows a program to execute multiple tasks or threads at the same time. By using resources more effectively, concurrency can enhance performance.

In Java, concurrency is achieved through *multithreading*, where separate threads operate independently but may still need to interact with shared resources.

**Real World Analogy: A Busy Restaurant**

Concurrency can be compared to the operations in a busy restaurant. In a restaurant, multiple customers are served at the same time by different waiters. Each waiter independently takes orders, serves food, and handles payments, helping ensure that every customer receives prompt attention.

**Key Points:**

**Multiple Tasks**: Just like multiple customers are served at once in a restaurant, a program can run several threads simultaneously, each performing a different task.

**Independent Workers**: Each waiter (representing a thread) operates independently, focusing on their assigned tables (tasks) without interfering with other waiters.

**Shared Resources**: Sometimes waiters need to access shared resources, like the kitchen, to fulfill orders. Similarly, threads in a program may need to share resources, such as memory or data, which can lead to potential issues like *race conditions* if not carefully managed.

### 

### 

### 

### 

### 

### **What is a Thread?**

### A thread is the smallest unit of a process (Sub-Process) that can be executed independently. It allows multiple tasks to be performed simultaneously within a program (concurrent execution). A program that runs a single thread is called a **single-threaded** program, while a program with multiple threads is a **multi-threaded** program.

### In simpler terms, think of a thread as a **line of work or task** that can operate separately but within the main application, allowing multiple tasks to be executed at the same time.

### **Real-World Analogy: Kitchen**

### Imagine you’re at a restaurant. There’s one main *kitchen* where food is prepared (representing the main application). However, the kitchen has multiple *chefs* (threads) who can each handle different tasks like cooking, chopping, or serving at the same time. Each chef can operate independently to complete orders quicker, much like threads that work concurrently to complete tasks within a program.

### In the illustration above, a thread operates within a process. This means that when multiple threads are running, they may switch back and forth between each other, this process is known as *context switching*.

### 

In the illustration above, a thread (t’s) operates within a process. This means that when multiple threads are running, they may switch back and forth between each other, this process is known as *context switching*.

### **Why Threading?**

### Threading is used to improve performance and responsiveness in applications. It allows the program to perform multiple operations at the same time without waiting for one operation to finish before starting another.

### **Benefits**:

### **Improved performance**: Tasks are executed in parallel, reducing overall execution time.

### **Better resource utilization**: Threads share the same memory, which is more efficient than creating separate processes.

### **Responsiveness**: For GUI applications, threading ensures that the UI does not freeze while background operations are performed.

### **Example**: In a text editor, threading allows you to keep typing while the program auto-saves the document in the background.

### 

### 

### 

### **What Is Multithreading**

### Multithreading is a Java feature that allows multiple threads to be executed simultaneously within a single program. This helps in performing multiple tasks concurrently, thereby enhancing efficiency, speed, and resource utilization.

### In simple terms, multithreading lets different parts of a program run at the same time, which makes the program faster and more responsive.

### 

### **Real-World Analogy**

### Imagine a **hotel** with multiple rooms and different tasks to manage, such as checking in guests, delivering room service, and cleaning. Each of these tasks can happen at the same time without interference.

### Similarly, in a multithreaded program, tasks (threads) can run in parallel, managed by a single application.

### 

### **Why Use Multithreading?**

### Multithreading has many benefits:

### **Increased Efficiency**: Multiple threads can complete tasks faster by running in parallel.

### **Improved Application Responsiveness**: Especially useful for applications that perform background operations, like file downloading.

### **Resource Optimization**: Maximizes CPU usage, as each thread utilizes a CPU core.

### **Reduced Downtime**: Since multiple tasks run together, the program doesn’t wait for one task to finish before starting another.

### **Now let’s see how to create a Thread**

### **Explanation**: In Java, threads can be created by extending the Thread class. This is one way to create a thread

### **Creating a Thread with the Thread class**

### Example Code

### ****class MyThread extends Thread {

### public void run() {

### System.out.println("Thread is running using Thread class");

### }

### }

### 

### public class TestThread {

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

### MyThread t1 = new MyThread();

### t1.start(); // Starts the thread, calling the run() method

### }

### }

### ****

### 

### 

### **start() Method**

### **Purpose**: The start() method is used to begin the execution of a thread. When you call start(), a new thread is created, and the run() method within that thread begins executing.

### **Why It's Used**: Without start(), a thread wouldn’t actually begin; it would simply exist without doing any work.

### **Analogy**: Imagine turning the ignition key of a car; this action "starts" the engine, allowing the car to move. Similarly, calling start() allows the thread to begin its execution.

### **Real-World Use**: This is commonly used whenever we want to execute multiple tasks concurrently, such as downloading files while updating a user interface.

### 

### **run() Method**

### **Purpose**: The run() method contains the code that is executed by a thread once it starts. This is where you put the actual work or logic that the thread will perform.

### **Why It's Used**: The run() method acts as the entry point for any actions the thread will take. While calling run() directly doesn’t start a new thread, the method itself is vital for defining what the thread will do.

### **Analogy**: Think of run() as a to-do list for a worker. The list contains all tasks the worker (thread) should do when assigned a job.

**Real-World Use**: Used in any thread-based application. For example, in a chat app, run() methods could handle listening for incoming messages in separate threads.

### 

### **Thread Life Cycle:**

### **New:** When the thread is created but not started.

### **Runnable:** After calling start(), the thread is ready to be executed.

### **Running:** The thread is actively executing the task.

### **Blocked/Waiting:** The thread is temporarily inactive and waits for some

### conditions or resources.

### **Terminated:** The thread finishes executing.

A thread progresses through several stages in its lifespan: birth, initiation, execution, and termination. The diagram illustrates this complete lifecycle.

