# Concurrency

The Java platform is designed from the ground up to support concurrent programming, with basic concurrency support in the Java programming language and the Java class libraries. Since version 5.0, the Java platform has also included high-level concurrency APIs. This lesson introduces the platform's basic concurrency support and summarizes some of the high-level APIs in the java.util.concurrent packages.

# Processes and Threads

In concurrent programming, there are two basic units of execution: *processes* and *threads*. In the Java programming language, concurrent programming is mostly concerned with threads. However, processes are also important.

A computer system normally has many active processes and threads. This is true even in systems that only have a single execution core, and thus only have one thread actually executing at any given moment. Processing time for a single core is shared among processes and threads through an OS feature called time slicing.

It's becoming more and more common for computer systems to have multiple processors or processors with multiple execution cores. This greatly enhances a system's capacity for concurrent execution of processes and threads — but concurrency is possible even on simple systems, without multiple processors or execution cores.

## Processes

A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.

Processes are often seen as synonymous with programs or applications. However, what the user sees as a single application may in fact be a set of cooperating processes. To facilitate communication between processes, most operating systems support *Inter Process Communication* (IPC) resources, such as pipes and sockets. IPC is used not just for communication between processes on the same system, but processes on different systems.

Most implementations of the Java virtual machine run as a single process. A Java application can create additional processes using a [ProcessBuilder](https://docs.oracle.com/javase/8/docs/api/java/lang/ProcessBuilder.html" \t "https://docs.oracle.com/javase/tutorial/essential/concurrency/_blank) object. Multiprocess applications are beyond the scope of this lesson.

## Threads

Threads are sometimes called *lightweight processes*. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.

Threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

Multithreaded execution is an essential feature of the Java platform. Every application has at least one thread — or several, if you count "system" threads that do things like memory management and signal handling. But from the application programmer's point of view, you start with just one thread, called the *main thread*. This thread has the ability to create additional threads, as we'll demonstrate in the next section.

# Defining and Starting a Thread

An application that creates an instance of Thread must provide the code that will run in that thread. There are two ways to do this:

* Provide a Runnable object. The [Runnable](https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html" \t "https://docs.oracle.com/javase/tutorial/essential/concurrency/_blank) interface defines a single method, run, meant to contain the code executed in the thread. The Runnable object is passed to the Thread constructor, as in the [HelloRunnable](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/HelloRunnable.java" \t "https://docs.oracle.com/javase/tutorial/essential/concurrency/_blank) example:

public class HelloRunnable implements Runnable {

public void run() {

System.out.println("Hello from a thread!");

}

public static void main(String args[]) {

(new Thread(new HelloRunnable())).start();

}

}

* Subclass Thread. The Thread class itself implements Runnable, though its run method does nothing. An application can subclass Thread, providing its own implementation of run, as in the [HelloThread](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/HelloThread.java" \t "https://docs.oracle.com/javase/tutorial/essential/concurrency/_blank) example:

public class HelloThread extends Thread {

public void run() {

System.out.println("Hello from a thread!");

}

public static void main(String args[]) {

(new HelloThread()).start();

}

}

Notice that both examples invoke Thread.start in order to start the new thread.

Which of these idioms should you use? The first idiom, which employs a Runnable object, is more general, because the Runnable object can subclass a class other than Thread. The second idiom is easier to use in simple applications, but is limited by the fact that your task class must be a descendant of Thread. This lesson focuses on the first approach, which separates the Runnable task from the Thread object that executes the task. Not only is this approach more flexible, but it is applicable to the high-level thread management APIs covered later.

# Synchronization

Threads communicate primarily by sharing access to fields and the objects reference fields refer to. This form of communication is extremely efficient, but makes two kinds of errors possible: *thread interference* and *memory consistency errors*. The tool needed to prevent these errors is *synchronization*.

However, synchronization can introduce thread contention, which occurs when two or more threads try to access the same resource simultaneously and cause the Java runtime to execute one or more threads more slowly, or even suspend their execution. [Starvation and livelock](https://docs.oracle.com/javase/tutorial/essential/concurrency/starvelive.html" \t "https://docs.oracle.com/javase/tutorial/essential/concurrency/_top) are forms of thread contention. See the section [Liveness](https://docs.oracle.com/javase/tutorial/essential/concurrency/liveness.html" \t "https://docs.oracle.com/javase/tutorial/essential/concurrency/_top) for more information.

This section covers the following topics:

* [Thread Interference](https://docs.oracle.com/javase/tutorial/essential/concurrency/interfere.html) describes how errors are introduced when multiple threads access shared data.
* [Memory Consistency Errors](https://docs.oracle.com/javase/tutorial/essential/concurrency/memconsist.html) describes errors that result from inconsistent views of shared memory.
* [Synchronized Methods](https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html) describes a simple idiom that can effectively prevent thread interference and memory consistency errors.
* [Implicit Locks and Synchronization](https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html) describes a more general synchronization idiom, and describes how synchronization is based on implicit locks.
* [Atomic Access](https://docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html) talks about the general idea of operations that can't be interfered with by other threads.

## Thread vs Process

1) A program in execution is often referred as process. A thread is a subset(part) of the process.

2) A process consists of multiple threads. A thread is a smallest part of the process that can execute concurrently with other parts(threads) of the process.

3) A process is sometime referred as task. A thread is often referred as lightweight process.

4) A process has its own address space. A thread uses the process’s address space and share it with the other threads of that process.