**THREAD IN JAVA**

**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) 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) 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) 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) 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.

The Thread class defines a number of methods useful for thread management. These include static methods, which provide information about, or affect the status of, the thread invoking the method. The other methods are invoked from other threads involved in managing the thread and Thread object. We'll examine some of these methods in the following sections.

# Pausing Execution with Sleep

Thread.sleep causes the current thread to suspend execution for a specified period. This is an efficient means of making processor time available to the other threads of an application or other applications that might be running on a computer system. The sleep method can also be used for pacing, as shown in the example that follows, and waiting for another thread with duties that are understood to have time requirements, as with the SimpleThreads example in a later section.

Two overloaded versions of sleep are provided: one that specifies the sleep time to the millisecond and one that specifies the sleep time to the nanosecond. However, these sleep times are not guaranteed to be precise, because they are limited by the facilities provided by the underlying OS. Also, the sleep period can be terminated by interrupts, as we'll see in a later section. In any case, you cannot assume that invoking sleep will suspend the thread for precisely the time period specified.

The [SleepMessages](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/SleepMessages.java) example uses sleep to print messages at four-second intervals:

public class SleepMessages {

public static void main(String args[])

throws InterruptedException {

String importantInfo[] = {

"Mares eat oats",

"Does eat oats",

"Little lambs eat ivy",

"A kid will eat ivy too"

};

for (int i = 0;

i < importantInfo.length;

i++) {

//Pause for 4 seconds

Thread.sleep(4000);

//Print a message

System.out.println(importantInfo[i]);

}

}

}

Notice that main declares that it throws InterruptedException. This is an exception that sleep throws when another thread interrupts the current thread while sleep is active. Since this application has not defined another thread to cause the interrupt, it doesn't bother to catch InterruptedException.

# Interrupts

An *interrupt* is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate. This is the usage emphasized in this lesson.

A thread sends an interrupt by invoking [interrupt](https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html#interrupt--) on the Thread object for the thread to be interrupted. For the interrupt mechanism to work correctly, the interrupted thread must support its own interruption.

## Supporting Interruption

How does a thread support its own interruption? This depends on what it's currently doing. If the thread is frequently invoking methods that throw InterruptedException, it simply returns from the run method after it catches that exception. For example, suppose the central message loop in the SleepMessages example were in the run method of a thread's Runnable object. Then it might be modified as follows to support interrupts:

for (int i = 0; i < importantInfo.length; i++) {

// Pause for 4 seconds

try {

Thread.sleep(4000);

} catch (InterruptedException e) {

// We've been interrupted: no more messages.

return;

}

// Print a message

System.out.println(importantInfo[i]);

}

Many methods that throw InterruptedException, such as sleep, are designed to cancel their current operation and return immediately when an interrupt is received.

What if a thread goes a long time without invoking a method that throws InterruptedException? Then it must periodically invoke Thread.interrupted, which returns true if an interrupt has been received. For example:

for (int i = 0; i < inputs.length; i++) {

heavyCrunch(inputs[i]);

if (Thread.interrupted()) {

// We've been interrupted: no more crunching.

return;

}

}

In this simple example, the code simply tests for the interrupt and exits the thread if one has been received. In more complex applications, it might make more sense to throw an InterruptedException:

if (Thread.interrupted()) {

throw new InterruptedException();

}

This allows interrupt handling code to be centralized in a catch clause.

## The Interrupt Status Flag

The interrupt mechanism is implemented using an internal flag known as the *interrupt status*. Invoking Thread.interrupt sets this flag. When a thread checks for an interrupt by invoking the static method Thread.interrupted, interrupt status is cleared. The non-static isInterrupted method, which is used by one thread to query the interrupt status of another, does not change the interrupt status flag.

By convention, any method that exits by throwing an InterruptedException clears interrupt status when it does so. However, it's always possible that interrupt status will immediately be set again, by another thread invoking interrupt.

# Joins

The join method allows one thread to wait for the completion of another. If t is a Thread object whose thread is currently executing,

t.join();

causes the current thread to pause execution until t's thread terminates. Overloads of join allow the programmer to specify a waiting period. However, as with sleep, join is dependent on the OS for timing, so you should not assume that join will wait exactly as long as you specify.

Like sleep, join responds to an interrupt by exiting with an InterruptedException.

# The SimpleThreads Example

The following example brings together some of the concepts of this section. [SimpleThreads](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/SimpleThreads.java) consists of two threads. The first is the main thread that every Java application has. The main thread creates a new thread from the Runnable object, MessageLoop, and waits for it to finish. If the MessageLoop thread takes too long to finish, the main thread interrupts it.

The MessageLoop thread prints out a series of messages. If interrupted before it has printed all its messages, the MessageLoop thread prints a message and exits.

public class SimpleThreads {

// Display a message, preceded by

// the name of the current thread

static void threadMessage(String message) {

String threadName =

Thread.currentThread().getName();

System.out.format("%s: %s%n",

threadName,

message);

}

private static class MessageLoop

implements Runnable {

public void run() {

String importantInfo[] = {

"Mares eat oats",

"Does eat oats",

"Little lambs eat ivy",

"A kid will eat ivy too"

};

try {

for (int i = 0;

i < importantInfo.length;

i++) {

// Pause for 4 seconds

Thread.sleep(4000);

// Print a message

threadMessage(importantInfo[i]);

}

} catch (InterruptedException e) {

threadMessage("I wasn't done!");

}

}

}

public static void main(String args[])

throws InterruptedException {

// Delay, in milliseconds before

// we interrupt MessageLoop

// thread (default one hour).

long patience = 1000 \* 60 \* 60;

// If command line argument

// present, gives patience

// in seconds.

if (args.length > 0) {

try {

patience = Long.parseLong(args[0]) \* 1000;

} catch (NumberFormatException e) {

System.err.println("Argument must be an integer.");

System.exit(1);

}

}

threadMessage("Starting MessageLoop thread");

long startTime = System.currentTimeMillis();

Thread t = new Thread(new MessageLoop());

t.start();

threadMessage("Waiting for MessageLoop thread to finish");

// loop until MessageLoop

// thread exits

while (t.isAlive()) {

threadMessage("Still waiting...");

// Wait maximum of 1 second

// for MessageLoop thread

// to finish.

t.join(1000);

if (((System.currentTimeMillis() - startTime) > patience)

&& t.isAlive()) {

threadMessage("Tired of waiting!");

t.interrupt();

// Shouldn't be long now

// -- wait indefinitely

t.join();

}

}

threadMessage("Finally!");

}

}

Link: [Processes and Threads (The Java™ Tutorials > Essential Classes > Concurrency) (oracle.com)](https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html)

**Table different between Process and Thread**

| **S.NO** | **Process** | **Thread** |
| --- | --- | --- |
| 1. | Process means any program is in execution. | Thread means segment of a process. |
| 2. | Process takes more time to terminate. | Thread takes less time to terminate. |
| 3. | It takes more time for creation. | It takes less time for creation. |
| 4. | It also takes more time for context switching. | It takes less time for context switching. |
| 5. | Process is less efficient in term of communication. | Thread is more efficient in term of communication. |
| 6. | Process consume more resources. | Thread consume less resources. |
| 7. | Process is isolated. | Threads share memory. |
| 8. | Process is called heavy weight process. | Thread is called light weight process. |
| 9. | Process switching uses interface in operating system. | Thread switching does not require to call a operating system and cause an interrupt to the kernel. |
| 10. | If one process is blocked then it will not effect the execution of other process | Second thread in the same task could not run, while one server thread is blocked. |
| 11. | Process has its own Process Control Block, Stack and Address Space. | Thread has Parents’ PCB, its own Thread Control Block and Stack and common Address space. |