

Module [java.base](#)

Package [java.lang](#)

Class Thread

java.lang.Thread

All Implemented Interfaces: [Runnable](#)

Direct Known Subclasses: [ForkJoinWorkerThread](#)

Since: 1.0

public class **Thread** extends [Object](#) implements [Runnable](#)

A *thread* is a thread of execution in a program.

The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not also be marked as a daemon. When code running in some thread creates a new **Thread** object, the new thread has its priority initially set equal to the priority of the creating thread, and is a daemon thread if and only if the creating thread is a daemon.

When a Java Virtual Machine starts up, there is usually a single non-daemon thread (which typically calls the method named `main` of some designated class).

The Java Virtual Machine continues to execute threads until either of the following occurs:

- The `exit` method of class `Runtime` has been called and the security manager has permitted the exit operation to take place.
- All threads that are not daemon threads have died, either by returning from the call to the `run` method or by throwing an exception that propagates beyond the `run` method.

There are two ways to create a new thread of execution.

1. One is to declare a class to be a subclass of Thread. This subclass should override the run method of class Thread. An instance of the subclass can then be allocated and started.
2. The other way to create a thread is to declare a class that implements the Runnable interface. That class then implements the run method. An instance of the class can then be allocated, passed as an argument when creating Thread, and started.

Every thread has a name for identification purposes. More than one thread may have the same name. If a name is not specified when a thread is created, a new name is generated for it.

Nested Class Summary

1. **static class [Thread.State](#):** - A thread state.
2. **static interface [Thread.UncaughtExceptionHandler](#):** - Interface for handlers invoked when a Thread abruptly terminates due to an uncaught exception.

Field Summary

1. **public static final int [MAX_PRIORITY](#):** - The maximum priority that a thread can have. **Value=10**
2. **public static final int [MIN_PRIORITY](#):** - The minimum priority that a thread can have. **Value= 1**
3. **public static final int [NORM_PRIORITY](#):** - The default priority that is assigned to a thread. **Value= 5**

Method Details

```
public static Thread currentThread()
```

Returns a reference to the currently executing thread object.

public static void yield ()

A hint to the scheduler that the current thread is willing to **give up** its current use of a processor. The scheduler is free to ignore this hint.

Yield is a heuristic attempt to improve relative progression between threads that would otherwise over-utilize a CPU. Its use should be combined with detailed profiling and benchmarking to ensure that it actually has the desired effect.

It is rarely appropriate to use this method. It may be useful for debugging or testing purposes, where it may help to reproduce bugs due to race conditions. It may also be useful when designing concurrency control constructs such as the ones in the [java.util.concurrent.locks](#) package.

public static void sleep (long millis) throws [InterruptedException](#)

Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

Subject to the precision and accuracy of system timers and schedulers. The thread does not lose ownership of any monitors.

Throws:

- [IllegalArgumentException](#) - if the value of millis is negative
- [InterruptedException](#) - if any thread has interrupted the current thread. The *interrupted status* of the current thread is cleared when this exception is thrown.

public static void sleep (long millis, int nanos) throws [InterruptedException](#)

Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds plus the specified number of nanoseconds, subject to the precision and accuracy of system timers and schedulers. The thread does not lose ownership of any monitors.

Throws:

- [IllegalArgumentException](#) - if the value of millis is negative, or the value of nanos is not in the range 0-999999
- [InterruptedException](#) - if any thread has interrupted the current thread. The *interrupted status* of the current thread is cleared when this exception is thrown.

public static void onSpinWait()

Since: 9

Indicates that the caller is momentarily unable to progress, until the occurrence of one or more actions on the part of other activities. By invoking this method within each iteration of a spin-wait loop construct, the calling thread indicates to the runtime that it is busy-waiting. The runtime may take action to improve the performance of invoking spin-wait loop constructions.

protected [Object](#) clone() throws [CloneNotSupportedException](#)

Throws **CloneNotSupportedException** as a Thread cannot be meaningfully cloned. Construct a new Thread instead.

public void start()

Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.

The result is that two threads are running concurrently: the current thread (which returns from the call to the start method) and the other thread (which executes its run method).

It is never legal to start a thread more than once. In particular, a thread may not be restarted once it has completed execution.

Throws: [IllegalThreadStateException](#) - if the thread was already started.

public void run()

If this thread was constructed using a separate `Runnable` run object, then that `Runnable` object's `run` method is called; otherwise, this method does nothing and returns.

Subclasses of `Thread` should override this method.

@Deprecated(since="1.2") public final void stop()

Deprecated.

Forces the thread to stop executing.

If there is a security manager installed, its `checkAccess` method is called with `this` as its argument. This may result in a `SecurityException` being raised (in the current thread).

If this thread is different from the current thread (that is, the current thread is trying to stop a thread other than itself), the security manager's `checkPermission` method (with a `RuntimePermission("stopThread")` argument) is called in addition. Again, this may result in throwing a `SecurityException` (in the current thread).

The thread represented by this thread is forced to stop whatever it is doing abnormally and to throw a newly created `ThreadDeath` object as an exception.

It is permitted to stop a thread that has not yet been started. If the thread is eventually started, it immediately terminates.

An application should not normally try to catch `ThreadDeath` unless it must do some extraordinary cleanup operation (note that the throwing of `ThreadDeath` causes `finally` clauses of `try` statements to be executed before the thread officially dies). If a `catch` clause catches a `ThreadDeath` object, it is important to rethrow the object so that the thread actually dies.

The top-level error handler that reacts to otherwise uncaught exceptions does not print out a message or otherwise notify the application if the uncaught exception is an instance of `ThreadDeath`.

Throws: [SecurityException](#) - if the current thread cannot modify this thread.

This method is inherently unsafe. Stopping a thread with `Thread.stop` causes it to unlock all of the monitors that it has locked (as a natural consequence of the unchecked `ThreadDeath` exception propagating up the stack). If any of the objects previously protected by these monitors were in an inconsistent state, the damaged objects become visible to other threads, potentially resulting in arbitrary behavior. Many uses of `stop` should be replaced by code that simply modifies some variable to indicate that the target thread should stop running. The target thread should check this variable regularly, and return from its run method in an orderly fashion if the variable indicates that it is to stop running. If the target thread waits for long periods (on a condition variable, for example), the `interrupt` method should be used to interrupt the wait.

public void interrupt()

Interrupts this thread.

Unless the current thread is interrupting itself, which is always permitted, the [checkAccess](#) method of this thread is invoked, which may cause a [SecurityException](#) to be thrown.

If this thread is blocked in an invocation of the [wait\(\)](#), [wait\(long\)](#), or [wait\(long, int\)](#) methods of the [Object](#) class, or of the [join\(\)](#), [join\(long\)](#), [join\(long, int\)](#), [sleep\(long\)](#), or [sleep\(long, int\)](#) methods of this class, then its interrupt status will be cleared and it will receive an [InterruptedException](#).

If this thread is blocked in an I/O operation upon an [InterruptibleChannel](#) then the channel will be closed, the thread's interrupt status will be set, and the thread will receive a [ClosedByInterruptException](#).

If this thread is blocked in a [Selector](#) then the thread's interrupt status will be set and it will return immediately from the selection operation, possibly with a non-zero value, just as if the selector's [wakeup](#) method were invoked.

If none of the previous conditions hold then this thread's interrupt status will be set.

Interrupting a thread that is not alive need not have any effect.

Implementation Note:

In the JDK Reference Implementation, interruption of a thread that is not alive still records that the interrupt request was made and will report it via `interrupted` and [`isInterrupted\(\)`](#).

Throws:

[`SecurityException`](#) - if the current thread cannot modify this thread

`public static boolean interrupted()`

Tests whether the current thread has been interrupted. The *interrupted status* of the thread is cleared by this method. In other words, if this method were to be called twice in succession, the second call would return false (unless the current thread were interrupted again, after the first call had cleared its interrupted status and before the second call had examined it).

Returns: true if the current thread has been interrupted; false otherwise.

`public boolean isInterrupted()`

Tests whether this thread has been interrupted. The *interrupted status* of the thread is unaffected by this method.

Returns: true if this thread has been interrupted; false otherwise.

`public final boolean isAlive()`

Tests if this thread is alive. A thread is alive if it has been started and has not yet died.

Returns: true if this thread is alive; false otherwise.

`@Deprecated(since="1.2", forRemoval=true) public final void suspend()`

Suspends this thread.

First, the `checkAccess` method of this thread is called with no arguments. This may result in throwing a `SecurityException` (in the current thread).

If the thread is alive, it is suspended and makes no further progress unless and until it is resumed.

Throws: [SecurityException](#) - if the current thread cannot modify this thread.

Deprecated, for removal: This API element is subject to removal in a future version.

This method has been deprecated, as it is inherently deadlock-prone. If the target thread holds a lock on the monitor protecting a critical system resource when it is suspended, no thread can access this resource until the target thread is resumed. If the thread that would resume the target thread attempts to lock this monitor prior to calling `resume`, deadlock results. Such deadlocks typically manifest themselves as "frozen" processes.

`@Deprecated(since="1.2", forRemoval=true) public final void resume()`

Resumes a suspended thread.

First, the `checkAccess` method of this thread is called with no arguments. This may result in throwing a `SecurityException` (in the current thread).

If the thread is alive but suspended, it is resumed and is permitted to make progress in its execution.

Throws: [SecurityException](#) - if the current thread cannot modify this thread.

Deprecated, for removal: This API element is subject to removal in a future version.

This method exists solely for use with [suspend\(\)](#), which has been deprecated because it is deadlock-prone.

`public final void setPriority (int newPriority)`

Changes the priority of this thread.

First the `checkAccess` method of this thread is called with no arguments. This may result in throwing a `SecurityException`.

Otherwise, the priority of this thread is set to the smaller of the specified `newPriority` and the maximum permitted priority of the thread's thread group.

Throws:

- [IllegalArgumentOutOfRangeException](#) - If the priority is not in the range `MIN_PRIORITY` to `MAX_PRIORITY`.
- [SecurityException](#) - if the current thread cannot modify this thread.

`public final int getPriority()`

Returns this thread's priority.

`public final void setName (String name)`

Changes the name of this thread to be equal to the argument name.

First the `checkAccess` method of this thread is called with no arguments. This may result in throwing a `SecurityException`.

Throws: [SecurityException](#) - if the current thread cannot modify this thread.

`public final String getName()`

Returns this thread's name.

`public final ThreadGroup getThreadGroup()`

Returns the thread group to which this thread belongs. This method returns null if this thread has died (been stopped).

`public static int activeCount()`

Returns an estimate of the number of active threads in the current thread's [thread group](#) and its subgroups. Recursively iterates over all subgroups in the current thread's thread group.

The value returned is only an estimate because the number of threads may change dynamically while this method traverses internal data structures, and might be affected by the presence of certain system threads. This method is intended primarily for debugging and monitoring purposes.

Returns: an estimate of the number of active threads in the current thread's thread group and in any other thread group that has the current thread's thread group as an ancestor

public static int enumerate ([Thread\[\]](#) tarray)

Copies into the specified array every active thread in the current thread's thread group and its subgroups. This method simply invokes the [ThreadGroup.enumerate\(Thread\[\]\)](#) method of the current thread's thread group.

An application might use the [activeCount](#) method to get an estimate of how big the array should be, however *if the array is too short to hold all the threads, the extra threads are silently ignored*. If it is critical to obtain every active thread in the current thread's thread group and its subgroups, the invoker should verify that the returned int value is strictly less than the length of tarray.

Due to the inherent race condition in this method, it is recommended that the method only be used for debugging and monitoring purposes.

Parameters: tarray - an array into which to put the list of threads

Returns: the number of threads put into the array

Throws: [SecurityException](#) - if [ThreadGroup.checkAccess\(\)](#) determines that the current thread cannot access its thread group

**[@Deprecated\(since="1.2", forRemoval=true\)](#) public int
countStackFrames()**

Throws UnsupportedOperationException.

Deprecated, for removal: This API element is subject to removal in a future version.

This method was originally designed to count the number of stack frames but the results were never well-defined and it depended on thread-suspension. This method is subject to removal in a future version of Java SE.

**public final void join (long millis) throws
[InterruptedException](#)**

Waits at most millis milliseconds for this thread to die. A timeout of 0 means to wait forever.

This implementation uses a loop of this.wait calls conditioned on this.isAlive. As a thread terminates the this.notifyAll method is invoked. It is recommended that applications not use wait, notify, or notifyAll on Thread instances.

Throws:

- [IllegalArgumentOutOfRangeException](#) - if the value of millis is negative
- [InterruptedException](#) - if any thread has interrupted the current thread. The *interrupted status* of the current thread is cleared when this exception is thrown.

**public final void join (long millis, int nanos) throws
[InterruptedException](#)**

Waits at most millis milliseconds plus nanos nanoseconds for this thread to die. If both arguments are 0, it means to wait forever.

This implementation uses a loop of this.wait calls conditioned on this.isAlive. As a thread terminates the this.notifyAll method is invoked. It is recommended that applications not use wait, notify, or notifyAll on Thread instances.

Throws:

- [IllegalArgumentOutOfRangeException](#) - if the value of millis is negative, or the value of nanos is not in the range 0-999999

- [InterruptedException](#) - if any thread has interrupted the current thread. The *interrupted status* of the current thread is cleared when this exception is thrown.

public final void join() throws [InterruptedException](#)

Waits for this thread to die.

An invocation of this method behaves in exactly the same way as the invocation [join\(0\)](#)

Throws: [InterruptedException](#) - if any thread has interrupted the current thread. The *interrupted status* of the current thread is cleared when this exception is thrown.

public static void dumpStack()

Prints a stack trace of the current thread to the standard error stream. This method is used only for debugging.

public final void setDaemon (boolean on)

Marks this thread as either a [daemon](#) thread or a user thread. The Java Virtual Machine exits when the only threads running are all daemon threads.

This method must be invoked before the thread is started.

Parameters: on - if true, marks this thread as a daemon thread

Throws:

- [IllegalThreadStateException](#) - if this thread is [alive](#)
- [SecurityException](#) - if [checkAccess\(\)](#) determines that the current thread cannot modify this thread

public final boolean isDaemon()

Tests if this thread is a daemon thread.

public final void checkAccess()

Determines if the currently running thread has permission to modify this thread.

If there is a security manager, its `checkAccess` method is called with this thread as its argument. This may result in throwing a `SecurityException`.

Throws: [SecurityException](#) - if the current thread is not allowed to access this thread.

public [String](#) toString()

Returns a string representation of this thread, including the thread's name, priority, and thread group.

public [ClassLoader](#) getContextClassLoader()

Returns the context `ClassLoader` for this thread. The context `ClassLoader` is provided by the creator of the thread for use by code running in this thread when loading classes and resources. If not [set](#), the default is the `ClassLoader` context of the parent thread. The context `ClassLoader` of the primordial thread is typically set to the class loader used to load the application.

Returns: the context `ClassLoader` for this thread, or `null` indicating the system class loader (or, failing that, the bootstrap class loader)

Throws: [SecurityException](#) - if a security manager is present, and the caller's class loader is not `null` and is not the same as or an ancestor of the context class loader, and the caller does not have the [RuntimePermission](#)("getClassLoader")

public void setContextClassLoader ([ClassLoader](#) cl)

Sets the context `ClassLoader` for this `Thread`. The context `ClassLoader` can be set when a thread is created, and allows the creator of the thread to provide the

appropriate class loader, through `getContextClassLoader`, to code running in the thread when loading classes and resources.

If a security manager is present, its [checkPermission](#) method is invoked with a [RuntimePermission](#) ("setContextClassLoader") permission to see if setting the context `ClassLoader` is permitted.

Parameters: `cl` - the context `ClassLoader` for this `Thread`, or null indicating the system class loader (or, failing that, the bootstrap class loader)

Throws: [SecurityException](#) - if the current thread cannot set the context `ClassLoader`

public static boolean holdsLock ([Object](#) obj)

Returns `true` if and only if the current thread holds the monitor lock on the specified object.

This method is designed to allow a program to assert that the current thread already holds a specified lock:

```
assert Thread.holdsLock(obj);
```

Parameters: `obj` - the object on which to test lock ownership

Returns: `true` if the current thread holds the monitor lock on the specified object.

Throws: [NullPointerException](#) - if `obj` is null

Since: 1.4

public [StackTraceElement](#)[] getStackTrace()

Returns an array of stack trace elements representing the stack dump of this thread. This method will return a zero-length array if this thread has not started, has started but has not yet been scheduled to run by the system, or has terminated. If the returned array is of non-zero length then the first element of the array represents the top of the stack, which is the most recent method invocation in the sequence. The last element of the array represents the bottom of the stack, which is the least recent method invocation in the sequence.

If there is a security manager, and this thread is not the current thread, then the security manager's `checkPermission` method is called with a `RuntimePermission("getStackTrace")` permission to see if it's ok to get the stack trace.

Some virtual machines may, under some circumstances, omit one or more stack frames from the stack trace. In the extreme case, a virtual machine that has no stack trace information concerning this thread is permitted to return a zero-length array from this method.

Returns: an array of `StackTraceElement`, each represents one stack frame.

Throws: [SecurityException](#) - if a security manager exists and its `checkPermission` method doesn't allow getting the stack trace of thread.

Since: 1.5

```
public static Map<Thread, StackTraceElement[]>  
    getAllStackTraces()
```

Returns a map of stack traces for all live threads. The map keys are threads and each map value is an array of `StackTraceElement` that represents the stack dump of the corresponding `Thread`. The returned stack traces are in the format specified for the [getStackTrace](#) method.

The threads may be executing while this method is called. The stack trace of each thread only represents a snapshot and each stack trace may be obtained at different time. A zero-length array will be returned in the map value if the virtual machine has no stack trace information about a thread.

If there is a security manager, then the security manager's `checkPermission` method is called with a `RuntimePermission("getStackTrace")` permission as well as `RuntimePermission("modifyThreadGroup")` permission to see if it is ok to get the stack trace of all threads.

Returns: a `Map` from `Thread` to an array of `StackTraceElement` that represents the stack trace of the corresponding thread.

Throws: [SecurityException](#) - if a security manager exists and its `checkPermission` method doesn't allow getting the stack trace of thread.

Since: 1.5

public long getId()

Returns the identifier of this Thread. The thread ID is a positive long number generated when this thread was created. The thread ID is unique and remains unchanged during its lifetime. When a thread is terminated, this thread ID may be reused.

Returns: this thread's ID.

Since: 1.5

public [Thread.State](#) getState()

Returns the state of this thread. This method is designed for use in monitoring of the system state, not for synchronization control.

Returns: this thread's state.

Since: 1.5

public static void setDefaultUncaughtExceptionHandler ([Thread.UncaughtExceptionHandler](#) eh)

Set the default handler invoked when a thread abruptly terminates due to an uncaught exception, and no other handler has been defined for that thread.

Uncaught exception handling is controlled first by the thread, then by the thread's [ThreadGroup](#) object and finally by the default uncaught exception handler. If the thread does not have an explicit uncaught exception handler set, and the thread's thread group (including parent thread groups) does not specialize its `uncaughtException` method, then the default handler's `uncaughtException` method will be invoked.

By setting the default uncaught exception handler, an application can change the way in which uncaught exceptions are handled (such as logging to a specific device, or file) for those threads that would already accept whatever "default" behavior the system provided.

Note that the default uncaught exception handler should not usually defer to the thread's ThreadGroup object, as that could cause infinite recursion.

Parameters: eh - the object to use as the default uncaught exception handler. If null then there is no default handler.

Throws: [SecurityException](#) - if a security manager is present and it denies [RuntimePermission](#)("setDefaultUncaughtExceptionHandler")

Since: 1.5

**public static [Thread.UncaughtExceptionHandler](#)
[getDefaultUncaughtExceptionHandler\(\)](#)**

Returns the default handler invoked when a thread abruptly terminates due to an uncaught exception. If the returned value is null, there is no default.

Returns: the default uncaught exception handler for all threads

Since: 1.5

**public [Thread.UncaughtExceptionHandler](#)
[getUncaughtExceptionHandler\(\)](#)**

Returns the handler invoked when this thread abruptly terminates due to an uncaught exception. If this thread has not had an uncaught exception handler explicitly set then this thread's ThreadGroup object is returned, unless this thread has terminated, in which case null is returned.

Returns: the uncaught exception handler for this thread

Since: 1.5

**public void [setUncaughtExceptionHandler](#)
([Thread.UncaughtExceptionHandler](#) eh)**

Set the handler invoked when this thread abruptly terminates due to an uncaught exception.

A thread can take full control of how it responds to uncaught exceptions by having its uncaught exception handler explicitly set. If no such handler is set then the thread's `ThreadGroup` object acts as its handler.

Parameters: eh - the object to use as this thread's uncaught exception handler. If null then this thread has no explicit handler.

Throws: [SecurityException](#) - if the current thread is not allowed to modify this thread.

Since: 1.5