Module java.base

Package java.lang

Class Object

java.lang.Object

Since: 1.0

public class **Object**

Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class

---

Constructors: -

Object (): - Constructs a new object.

Method Summary: -

1. **protected Object clone (): -** Creates and returns a copy of this object.
2. **boolean equals (Object obj): -** Indicates whether some other object is "equal to" this one.
3. **protected void finalize (): -** The finalization mechanism is inherently problematic. **[Deprecated]**
4. **Class<?> getClass(): -**Returns the runtime class of this Object.
5. **Int hashCode(): -** Returns a hash code value for the object.
6. **void notify (): -** Wakes up a single thread that is waiting on this object's monitor.
7. **void notifyAll(): -** Wakes up all threads that are waiting on this object's monitor.
8. **void wait (): -** Causes the current thread to wait until it is awakened, typically by being notified or interrupted.
9. **void wait (long timeoutMillis): -** Causes the current thread to wait until it is awakened, typically by being notified or interrupted, or until a certain amount of real time has elapsed.

10. **void wait (long timeoutMillis, int nanos): -** Causes the current thread to wait until it is awakened, typically by being notified or interrupted, or until a certain amount of real time has elapsed.
11. **String toString(): -** Returns a string representation of the object.

hashCode

public int hashCode()

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by HashMap.

The general contract of hashCode is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the equals (Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

equals

public boolean equals (Object obj)

Indicates whether some other object is "equal to" this one.

The equals method implements an equivalence relation on non-null object references:

- **It is reflexive:** for any non-null reference value x, x.equals(x) should return true.
- **It is symmetric:** for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
- **It is transitive:** for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
- **It is consistent:** for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
- For any non-null reference value x, x.equals(null) should return false.

The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object (x == y has the value true).

Note that it is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

notify

public final void notify ()

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. A thread waits on an object's monitor by calling one of the wait methods.

The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object. The awakened thread will compete in the usual manner with any other threads that might be actively competing to synchronize on this object; for example, the awakened thread enjoys no reliable privilege or disadvantage in being the next thread to lock this object.

This method should only be called by a thread that is the owner of this object's monitor. A thread becomes the owner of the object's monitor in one of three ways:

- By executing a synchronized instance method of that object.
- By executing the body of a synchronized statement that synchronizes on the object.
- For objects of type Class, by executing a synchronized static method of that class.

Only one thread at a time can own an object's monitor.

Throws: IllegalMonitorStateException - if the current thread is not the owner of this object's monitor.

notifyAll

public final void notifyAll()

Wakes up all threads that are waiting on this object's monitor. A thread waits on an object's monitor by calling one of the wait methods.

The awakened threads will not be able to proceed until the current thread relinquishes the lock on this object. The awakened threads will compete in the usual manner with any other threads that might be actively competing to synchronize on this object; for example, the awakened threads enjoy no reliable privilege or disadvantage in being the next thread to lock this object.

This method should only be called by a thread that is the owner of this object's monitor. See the notify method for a description of the ways in which a thread can become the owner of a monitor.

Throws: IllegalMonitorStateException - if the current thread is not the owner of this object's monitor.

wait

public final void wait () throws InterruptedException

Causes the current thread to wait until it is awakened, typically by being notified or interrupted.

Throws: -

- **IllegalMonitorStateException -** if the current thread is not the owner of the object's monitor
- **InterruptedException -** if any thread interrupted the current thread before or while the current thread was waiting. The interrupted status of the current thread is cleared when this exception is thrown.

wait

public final void wait (long timeoutMillis) throws InterruptedException

Causes the current thread to wait until it is awakened, typically by being notified or interrupted, or until a certain amount of real time has elapsed.

Throws: -

- **IllegalArgumentException -** if timeoutMillis is negative
- **IllegalMonitorStateException -** if the current thread is not the owner of the object's monitor
- **InterruptedException -** if any thread interrupted the current thread before or while the current thread was waiting. The interrupted status of the current thread is cleared when this exception is thrown.

wait

public final void wait (long timeoutMillis, int nanos) throws InterruptedException

Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*, or until a certain amount of real time has elapsed.

The current thread must own this object's monitor lock.

This method causes the current thread (referred to here as **T**) to place itself in the wait set for this object and then to relinquish any and all synchronization claims on this object.

Note that only the locks on this object are relinquished; any other objects on which the current thread may be synchronized remain locked while the thread waits.

Thread **T** then becomes disabled for thread scheduling purposes and lies dormant until one of the following occurs:

- Some other thread invokes the notify method for this object and thread T happens to be arbitrarily chosen as the thread to be awakened.
- Some other thread invokes the notifyAll method for this object.
- Some other thread interrupts thread T.
- The specified amount of real time has elapsed, more or less. The amount of real time, in nanoseconds, is given by the expression 1000000 * timeoutMillis + nanos. If timeoutMillis and nanos are both zero, then real time is not taken into consideration and the thread waits until awakened by one of the other causes.
- Thread T is awakened spuriously. (See below.)

The thread T is then removed from the wait set for this object and re-enabled for thread scheduling. It competes in the usual manner with other threads for the right to synchronize on the object; once it has regained control of the object, all its synchronization claims on the object are restored to the status quo ante - that is, to the situation as of the time that the wait method was invoked. Thread T then returns from the invocation of the wait method. Thus, on return from the wait method, the synchronization state of the object and of thread T is exactly as it was when the wait method was invoked.

A thread can wake up without being notified, interrupted, or timing out, a so-called *spurious wakeup*. While this will rarely occur in practice, applications must guard against it by testing for the condition that should have caused the thread to be awakened, and continuing to wait if the condition is not satisfied. See the example below.

If the current thread is interrupted by any thread before or while it is waiting, then an InterruptedException is thrown. The *interrupted status* of the current thread is cleared when this exception is thrown. This exception is not thrown until the lock status of this object has been restored as described above.

API Note:

The recommended approach to waiting is to check the condition being awaited in a while loop around the call to wait, as shown in the example below. Among other things, this approach avoids problems that can be caused by spurious wakeups.

```
synchronized (obj) {
  while (<condition does not hold> and <timeout not exceeded>) {
      long timeoutMillis = ... ; // recompute timeout values
      int nanos = ... ;
      obj.wait(timeoutMillis, nanos);
       }
... // Perform action appropriate to condition or timeout     }
```

Throws:

- IllegalArgumentException - if timeoutMillis is negative, or if the value of nanos is out of range
- IllegalMonitorStateException - if the current thread is not the owner of the object's monitor
- InterruptedException - if any thread interrupted the current thread before or while the current thread was waiting. The *interrupted status* of the current thread is cleared when this exception is thrown.

finalize

@Deprecated(since="9")

protected void finalize() throws Throwable

Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.

A subclass overrides the finalize method to dispose of system resources or to perform other cleanup.

The general contract of finalize is that it is invoked if and when the Java virtual machine has determined that there is no longer any means by which this object can be accessed by any thread that has not yet died, except as a result of an action taken by the finalization of some other object or class which is ready to be finalized.

The finalize method may take any action, including making this object available again to other threads; the usual purpose of finalize, however, is to perform cleanup actions before the object is irrevocably discarded. For example, the finalize method for an object that represents an input/output connection might perform explicit I/O transactions to break the connection before the object is permanently discarded.

The finalize method of class Object performs no special action; it simply returns normally. Subclasses of Object may override this definition.

The Java programming language does not guarantee which thread will invoke the finalize method for any given object. It is guaranteed, however, that the thread that invokes finalize will not be holding any user-visible synchronization locks when finalize is invoked. If an uncaught exception is thrown by the finalize method, the exception is ignored and finalization of that object terminates.

After the finalize method has been invoked for an object, no further action is taken until the Java virtual machine has again determined that there is no longer any means by which this object can be accessed by any thread that has not yet died, including possible actions by other objects or classes which are ready to be finalized, at which point the object may be discarded.

The finalize method is never invoked more than once by a Java virtual machine for any given object.

Any exception thrown by the finalize method causes the finalization of this object to be halted, but is otherwise ignored.

Deprecated.

The finalization mechanism is inherently problematic. Finalization can lead to performance issues, deadlocks, and hangs. Errors in finalizers can lead to resource leaks; there is no way to cancel finalization if it is no longer necessary; and no ordering is specified among calls to finalize methods of different objects. Furthermore, there are no guarantees regarding the timing of finalization. The finalize method might be called on a finalizable object only after an indefinite delay, if at all. Classes whose instances hold non-heap resources should provide a method to enable explicit release of those resources, and they should also implement AutoCloseable if appropriate. The Cleaner and PhantomReference provide more flexible and efficient ways to release resources when an object becomes unreachable.