package java.lang;

/\*\*

\* Class {@code Object} is the root of the class hierarchy.

\* Every class has {@code Object} as a superclass. All objects,

\* including arrays, implement the methods of this class.

\*

\* @author unascribed

\* @see java.lang.Class

\* @since JDK1.0

\*/

public class Object {

private static native void registerNatives();

static {

registerNatives();

}

/\*\*

\* Returns the runtime class of this {@code Object}. The returned

\* {@code Class} object is the object that is locked by {@code

\* static synchronized} methods of the represented class.

\*

\* <p><b>The actual result type is {@code Class<? extends |X|>}

\* where {@code |X|} is the erasure of the static type of the

\* expression on which {@code getClass} is called.</b> For

\* example, no cast is required in this code fragment:</p>

\*

\* <p>

\* {@code Number n = 0; }<br>

\* {@code Class<? extends Number> c = n.getClass(); }

\* </p>

\*

\* @return The {@code Class} object that represents the runtime

\* class of this object.

\* @jls 15.8.2 Class Literals

\*/

public final native Class<?> getClass();

/\*\*

\* Returns a hash code value for the object. This method is

\* supported for the benefit of hash tables such as those provided by

\* {@link java.util.HashMap}.

\* <p>

\* The general contract of {@code hashCode} is:

\* <ul>

\* <li>Whenever it is invoked on the same object more than once during

\* an execution of a Java application, the {@code hashCode} method

\* must consistently return the same integer, provided no information

\* used in {@code 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.

\* <li>If two objects are equal according to the {@code equals(Object)}

\* method, then calling the {@code hashCode} method on each of

\* the two objects must produce the same integer result.

\* <li>It is <em>not</em> required that if two objects are unequal

\* according to the {@link java.lang.Object#equals(java.lang.Object)}

\* method, then calling the {@code 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.

\* </ul>

\* <p>

\* As much as is reasonably practical, the hashCode method defined by

\* class {@code Object} does return distinct integers for distinct

\* objects. (This is typically implemented by converting the internal

\* address of the object into an integer, but this implementation

\* technique is not required by the

\* Java&trade; programming language.)

\*

\* @return a hash code value for this object.

\* @see java.lang.Object#equals(java.lang.Object)

\* @see java.lang.System#identityHashCode

\*/

public native int hashCode();

/\*\*

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

\* <p>

\* The {@code equals} method implements an equivalence relation

\* on non-null object references:

\* <ul>

\* <li>It is <i>reflexive</i>: for any non-null reference value

\* {@code x}, {@code x.equals(x)} should return

\* {@code true}.

\* <li>It is <i>symmetric</i>: for any non-null reference values

\* {@code x} and {@code y}, {@code x.equals(y)}

\* should return {@code true} if and only if

\* {@code y.equals(x)} returns {@code true}.

\* <li>It is <i>transitive</i>: for any non-null reference values

\* {@code x}, {@code y}, and {@code z}, if

\* {@code x.equals(y)} returns {@code true} and

\* {@code y.equals(z)} returns {@code true}, then

\* {@code x.equals(z)} should return {@code true}.

\* <li>It is <i>consistent</i>: for any non-null reference values

\* {@code x} and {@code y}, multiple invocations of

\* {@code x.equals(y)} consistently return {@code true}

\* or consistently return {@code false}, provided no

\* information used in {@code equals} comparisons on the

\* objects is modified.

\* <li>For any non-null reference value {@code x},

\* {@code x.equals(null)} should return {@code false}.

\* </ul>

\* <p>

\* The {@code equals} method for class {@code Object} implements

\* the most discriminating possible equivalence relation on objects;

\* that is, for any non-null reference values {@code x} and

\* {@code y}, this method returns {@code true} if and only

\* if {@code x} and {@code y} refer to the same object

\* ({@code x == y} has the value {@code true}).

\* <p>

\* Note that it is generally necessary to override the {@code hashCode}

\* method whenever this method is overridden, so as to maintain the

\* general contract for the {@code hashCode} method, which states

\* that equal objects must have equal hash codes.

\*

\* @param obj the reference object with which to compare.

\* @return {@code true} if this object is the same as the obj

\* argument; {@code false} otherwise.

\* @see #hashCode()

\* @see java.util.HashMap

\*/

public boolean equals(Object obj) {

return (this == obj);

}

/\*\*

\* Creates and returns a copy of this object. The precise meaning

\* of "copy" may depend on the class of the object. The general

\* intent is that, for any object {@code x}, the expression:

\* <blockquote>

\* <pre>

\* x.clone() != x</pre></blockquote>

\* will be true, and that the expression:

\* <blockquote>

\* <pre>

\* x.clone().getClass() == x.getClass()</pre></blockquote>

\* will be {@code true}, but these are not absolute requirements.

\* While it is typically the case that:

\* <blockquote>

\* <pre>

\* x.clone().equals(x)</pre></blockquote>

\* will be {@code true}, this is not an absolute requirement.

\* <p>

\* By convention, the returned object should be obtained by calling

\* {@code super.clone}. If a class and all of its superclasses (except

\* {@code Object}) obey this convention, it will be the case that

\* {@code x.clone().getClass() == x.getClass()}.

\* <p>

\* By convention, the object returned by this method should be independent

\* of this object (which is being cloned). To achieve this independence,

\* it may be necessary to modify one or more fields of the object returned

\* by {@code super.clone} before returning it. Typically, this means

\* copying any mutable objects that comprise the internal "deep structure"

\* of the object being cloned and replacing the references to these

\* objects with references to the copies. If a class contains only

\* primitive fields or references to immutable objects, then it is usually

\* the case that no fields in the object returned by {@code super.clone}

\* need to be modified.

\* <p>

\* The method {@code clone} for class {@code Object} performs a

\* specific cloning operation. First, if the class of this object does

\* not implement the interface {@code Cloneable}, then a

\* {@code CloneNotSupportedException} is thrown. Note that all arrays

\* are considered to implement the interface {@code Cloneable} and that

\* the return type of the {@code clone} method of an array type {@code T[]}

\* is {@code T[]} where T is any reference or primitive type.

\* Otherwise, this method creates a new instance of the class of this

\* object and initializes all its fields with exactly the contents of

\* the corresponding fields of this object, as if by assignment; the

\* contents of the fields are not themselves cloned. Thus, this method

\* performs a "shallow copy" of this object, not a "deep copy" operation.

\* <p>

\* The class {@code Object} does not itself implement the interface

\* {@code Cloneable}, so calling the {@code clone} method on an object

\* whose class is {@code Object} will result in throwing an

\* exception at run time.

\*

\* @return a clone of this instance.

\* @throws CloneNotSupportedException if the object's class does not

\* support the {@code Cloneable} interface. Subclasses

\* that override the {@code clone} method can also

\* throw this exception to indicate that an instance cannot

\* be cloned.

\* @see java.lang.Cloneable

\*/

protected native Object clone() throws CloneNotSupportedException;

/\*\*

\* Returns a string representation of the object. In general, the

\* {@code toString} method returns a string that

\* "textually represents" this object. The result should

\* be a concise but informative representation that is easy for a

\* person to read.

\* It is recommended that all subclasses override this method.

\* <p>

\* The {@code toString} method for class {@code Object}

\* returns a string consisting of the name of the class of which the

\* object is an instance, the at-sign character `{@code @}', and

\* the unsigned hexadecimal representation of the hash code of the

\* object. In other words, this method returns a string equal to the

\* value of:

\* <blockquote>

\* <pre>

\* getClass().getName() + '@' + Integer.toHexString(hashCode())

\* </pre></blockquote>

\*

\* @return a string representation of the object.

\*/

public String toString() {

return getClass().getName() + "@" + Integer.toHexString(hashCode());

}

/\*\*

\* 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 {@code wait} methods.

\* <p>

\* 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.

\* <p>

\* 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:

\* <ul>

\* <li>By executing a synchronized instance method of that object.

\* <li>By executing the body of a {@code synchronized} statement

\* that synchronizes on the object.

\* <li>For objects of type {@code Class,} by executing a

\* synchronized static method of that class.

\* </ul>

\* <p>

\* 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.

\* @see java.lang.Object#notifyAll()

\* @see java.lang.Object#wait()

\*/

public final native void notify();

/\*\*

\* 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

\* {@code wait} methods.

\* <p>

\* 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.

\* <p>

\* This method should only be called by a thread that is the owner

\* of this object's monitor. See the {@code 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.

\* @see java.lang.Object#notify()

\* @see java.lang.Object#wait()

\*/

public final native void notifyAll();

/\*\*

\* Causes the current thread to wait until either another thread invokes the

\* {@link java.lang.Object#notify()} method or the

\* {@link java.lang.Object#notifyAll()} method for this object, or a

\* specified amount of time has elapsed.

\* <p>

\* The current thread must own this object's monitor.

\* <p>

\* This method causes the current thread (call it <var>T</var>) to

\* place itself in the wait set for this object and then to relinquish

\* any and all synchronization claims on this object. Thread <var>T</var>

\* becomes disabled for thread scheduling purposes and lies dormant

\* until one of four things happens:

\* <ul>

\* <li>Some other thread invokes the {@code notify} method for this

\* object and thread <var>T</var> happens to be arbitrarily chosen as

\* the thread to be awakened.

\* <li>Some other thread invokes the {@code notifyAll} method for this

\* object.

\* <li>Some other thread {@linkplain Thread#interrupt() interrupts}

\* thread <var>T</var>.

\* <li>The specified amount of real time has elapsed, more or less. If

\* {@code timeout} is zero, however, then real time is not taken into

\* consideration and the thread simply waits until notified.

\* </ul>

\* The thread <var>T</var> is then removed from the wait set for this

\* object and re-enabled for thread scheduling. It then competes in the

\* usual manner with other threads for the right to synchronize on the

\* object; once it has gained 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 {@code wait}

\* method was invoked. Thread <var>T</var> then returns from the

\* invocation of the {@code wait} method. Thus, on return from the

\* {@code wait} method, the synchronization state of the object and of

\* thread {@code T} is exactly as it was when the {@code wait} method

\* was invoked.

\* <p>

\* A thread can also wake up without being notified, interrupted, or

\* timing out, a so-called <i>spurious wakeup</i>. 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. In other words,

\* waits should always occur in loops, like this one:

\* <pre>

\* synchronized (obj) {

\* while (&lt;condition does not hold&gt;)

\* obj.wait(timeout);

\* ... // Perform action appropriate to condition

\* }

\* </pre>

\* (For more information on this topic, see Section 3.2.3 in Doug Lea's

\* "Concurrent Programming in Java (Second Edition)" (Addison-Wesley,

\* 2000), or Item 50 in Joshua Bloch's "Effective Java Programming

\* Language Guide" (Addison-Wesley, 2001).

\*

\* <p>If the current thread is {@linkplain java.lang.Thread#interrupt()

\* interrupted} by any thread before or while it is waiting, then an

\* {@code InterruptedException} is thrown. This exception is not

\* thrown until the lock status of this object has been restored as

\* described above.

\*

\* <p>

\* Note that the {@code wait} method, as it places the current thread

\* into the wait set for this object, unlocks only this object; any

\* other objects on which the current thread may be synchronized remain

\* locked while the thread waits.

\* <p>

\* This method should only be called by a thread that is the owner

\* of this object's monitor. See the {@code notify} method for a

\* description of the ways in which a thread can become the owner of

\* a monitor.

\*

\* @param timeout the maximum time to wait in milliseconds.

\* @throws IllegalArgumentException if the value of timeout is

\* negative.

\* @throws IllegalMonitorStateException if the current thread is not

\* the owner of the object's monitor.

\* @throws InterruptedException if any thread interrupted the

\* current thread before or while the current thread

\* was waiting for a notification. The <i>interrupted

\* status</i> of the current thread is cleared when

\* this exception is thrown.

\* @see java.lang.Object#notify()

\* @see java.lang.Object#notifyAll()

\*/

public final native void wait(long timeout) throws InterruptedException;

/\*\*

\* Causes the current thread to wait until another thread invokes the

\* {@link java.lang.Object#notify()} method or the

\* {@link java.lang.Object#notifyAll()} method for this object, or

\* some other thread interrupts the current thread, or a certain

\* amount of real time has elapsed.

\* <p>

\* This method is similar to the {@code wait} method of one

\* argument, but it allows finer control over the amount of time to

\* wait for a notification before giving up. The amount of real time,

\* measured in nanoseconds, is given by:

\* <blockquote>

\* <pre>

\* 1000000\*timeout+nanos</pre></blockquote>

\* <p>

\* In all other respects, this method does the same thing as the

\* method {@link #wait(long)} of one argument. In particular,

\* {@code wait(0, 0)} means the same thing as {@code wait(0)}.

\* <p>

\* The current thread must own this object's monitor. The thread

\* releases ownership of this monitor and waits until either of the

\* following two conditions has occurred:

\* <ul>

\* <li>Another thread notifies threads waiting on this object's monitor

\* to wake up either through a call to the {@code notify} method

\* or the {@code notifyAll} method.

\* <li>The timeout period, specified by {@code timeout}

\* milliseconds plus {@code nanos} nanoseconds arguments, has

\* elapsed.

\* </ul>

\* <p>

\* The thread then waits until it can re-obtain ownership of the

\* monitor and resumes execution.

\* <p>

\* As in the one argument version, interrupts and spurious wakeups are

\* possible, and this method should always be used in a loop:

\* <pre>

\* synchronized (obj) {

\* while (&lt;condition does not hold&gt;)

\* obj.wait(timeout, nanos);

\* ... // Perform action appropriate to condition

\* }

\* </pre>

\* This method should only be called by a thread that is the owner

\* of this object's monitor. See the {@code notify} method for a

\* description of the ways in which a thread can become the owner of

\* a monitor.

\*

\* @param timeout the maximum time to wait in milliseconds.

\* @param nanos additional time, in nanoseconds range

\* 0-999999.

\* @throws IllegalArgumentException if the value of timeout is

\* negative or the value of nanos is

\* not in the range 0-999999.

\* @throws IllegalMonitorStateException if the current thread is not

\* the owner of this object's monitor.

\* @throws InterruptedException if any thread interrupted the

\* current thread before or while the current thread

\* was waiting for a notification. The <i>interrupted

\* status</i> of the current thread is cleared when

\* this exception is thrown.

\*/

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

if (timeout < 0) {

throw new IllegalArgumentException("timeout value is negative");

}

if (nanos < 0 || nanos > 999999) {

throw new IllegalArgumentException(

"nanosecond timeout value out of range");

}

if (nanos > 0) {

timeout++;

}

wait(timeout);

}

/\*\*

\* Causes the current thread to wait until another thread invokes the

\* {@link java.lang.Object#notify()} method or the

\* {@link java.lang.Object#notifyAll()} method for this object.

\* In other words, this method behaves exactly as if it simply

\* performs the call {@code wait(0)}.

\* <p>

\* The current thread must own this object's monitor. The thread

\* releases ownership of this monitor and waits until another thread

\* notifies threads waiting on this object's monitor to wake up

\* either through a call to the {@code notify} method or the

\* {@code notifyAll} method. The thread then waits until it can

\* re-obtain ownership of the monitor and resumes execution.

\* <p>

\* As in the one argument version, interrupts and spurious wakeups are

\* possible, and this method should always be used in a loop:

\* <pre>

\* synchronized (obj) {

\* while (&lt;condition does not hold&gt;)

\* obj.wait();

\* ... // Perform action appropriate to condition

\* }

\* </pre>

\* This method should only be called by a thread that is the owner

\* of this object's monitor. See the {@code 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 the object's monitor.

\* @throws InterruptedException if any thread interrupted the

\* current thread before or while the current thread

\* was waiting for a notification. The <i>interrupted

\* status</i> of the current thread is cleared when

\* this exception is thrown.

\* @see java.lang.Object#notify()

\* @see java.lang.Object#notifyAll()

\*/

public final void wait() throws InterruptedException {

wait(0);

}

/\*\*

\* 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 {@code finalize} method to dispose of

\* system resources or to perform other cleanup.

\* <p>

\* The general contract of {@code finalize} is that it is invoked

\* if and when the Java&trade; 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 {@code finalize} method may take any action, including

\* making this object available again to other threads; the usual purpose

\* of {@code 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.

\* <p>

\* The {@code finalize} method of class {@code Object} performs no

\* special action; it simply returns normally. Subclasses of

\* {@code Object} may override this definition.

\* <p>

\* The Java programming language does not guarantee which thread will

\* invoke the {@code 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.

\* <p>

\* After the {@code 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.

\* <p>

\* The {@code finalize} method is never invoked more than once by a Java

\* virtual machine for any given object.

\* <p>

\* Any exception thrown by the {@code finalize} method causes

\* the finalization of this object to be halted, but is otherwise

\* ignored.

\*

\* @throws Throwable the {@code Exception} raised by this method

\* @see java.lang.ref.WeakReference

\* @see java.lang.ref.PhantomReference

\* @jls 12.6 Finalization of Class Instances

\*/

protected void finalize() throws Throwable { }

}