public final native Class<?> getClass();

This method returns the class in this form: "class NameOfRuntimeClass", if the class is inherited from another, then the method returns “class NameOfParentClass$NameOfRuntimeClass”. If there are a lot of parent classes, method **getClass()** returns all parent classes splited by “$”.

public native int hashCode();

When calling this method, it calculates and returns a specific code according to a certain algorithm, which will be the same for identical objects. It is possible that the hashcode will be the same for two different objects, but if the hashcode is different, then these objects will be exactly different.

public boolean equals(Object obj) {  
 return (this == obj);  
}

This method compares the contents of objects and displays a boolean value of true if the content is equivalent, and false if not. Also, the content should not be null, because the method will return false.

we pass to this method variable of type Object

protected native Object clone() throws CloneNotSupportedException;

When we call the **clone ()** method, Java checks if the object has a Cloneable interface. If yes, it clones the object using the **clone ()** method, which means that another object is created and the fields of the sample object are assigned to its fields, if not, it throws out a CloneNotSupportedException exception.

public String toString() {  
 return getClass().getName() + "@" + Integer.*toHexString*(hashCode());  
}

The method returns the value of the object as a String. It can return a set of meaningless characters, but in fact there is a certain logic in them. If the method returns a string of characters, then most likely we are trying to output an array, for it there is a special implementation through **Arrays.toString().**

public final native void notify();

Calling the **notify ()** method wakes up only one thread, after which this thread starts execution. If an object is waiting for multiple threads, then the **notify ()** method will wake up only one of them. The choice of thread depends on the system implementation of thread control. Also, as I understand this thread locks the runtime object.

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 (@code synchronized) statement that synchronizes on the object.
* For objects of type (@code Class) by executing 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.)

public final native void notifyAll();

The **notifyAll ()** method wakes up all the threads, although in what sequence they will wake up depends on the OS implementation. The awakened threads will not be able to proceed until the current thread turns off the lock on this object. The awakened threads will compete for the right to lock current object; for example, the awakened threads have no reliable privileges or advantages in being the next thread to lock this object.

* throws IllegalMonitorStateException (if the current thread is not the owner of this object's monitor.)

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

The **wait ()** method has three variations. One **wait ()** method waits endlessly for another thread until **notify ()** or **notifyAll ()** is called on an object. The other two variations of the **wait ()** method put the current thread on hold for a specific time. After this time, the thread wakes up and continues to work.

* In case we need to wake up this method after some time:
* We should put in this method *timeout* (the maximum time to wait in milliseconds).

It throws 3 Exceptions:

* IllegalArgumentException (if the value of timeout 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 for a notification. The “interrupted status” 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);  
  }

This method is similar to the wait(long timeout) method of one argument, but it allows finer control over the amount of time to wait for a notification before giving up.

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

* In case we need to wake up this method after some time :
  + We should put in this method: *timeout* (the maximum time to wait in milliseconds.)
* And we should put in this method: *nanos* (additional time, in nanoseconds range 0-999999.)
* This method throws IllegalArgumentException (if the value of timeout is negative or the value of nanos is not in the range 0-999999.)
* This method throws IllegalMonitorStateException (if the current thread is not the owner of this object's monitor. )
* This method throws InterruptedException (if any thread interrupted the current thread before or while the current thread was waiting for a notification. The “interrupted status” of the current thread is cleared when this exception is thrown.)
* public final void wait() throws InterruptedException {  
   wait(0);  
  }

In other words, this method behaves exactly as if it simply performs the call **wait(0)**.

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 **notify()** method or the **notifyAll()** method.

* This method throws IllegalMonitorStateException (if the current thread is not the owner of the object's monitor.)
* This method throws InterruptedException (if any thread interrupted the current thread before or while the current thread was waiting for a notification. The “interrupted status” of the current thread is cleared when this exception is thrown.)

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. We don’t know exactly when this method will be called. Using finalization, we can define specific actions that will be performed immediately before the object is deleted by the garbage collector. The normal operation of the program should not depend on the finalize () method, since this method is not reliable

* Calls to the finalize () method are prohibited in the application code;
* The finalize () method must have a protected access modifier;
* Implementation of the finalize () method should call the ancestor method.
* This method throws Throwable the (Exception} raised by this method