# Introduction

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 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.

## Creating Threads

A Java thread can be created in two ways: by extending the Thread class or by implementing the Runnable interface.Both of them have a method named run(). The JVM will call this method when a thread starts executing. You can think of the run() method as a starting point for the execution of a thread, just like the main() method, which is the starting point for the execution of a program. You’ll first see two examples for creating threads—extend Thread and implement Runnable—before learning the differences between them.

### Extending the Thread Class

You’ll first consider how to extend the Thread class. You need to override the run() method when you want to extend the Thread class. If you don’t override the run() method, the default run() method from the Thread class will be called, which does nothing. To override the run() method, you need to declare it as public; it takes no arguments and has a void return type—in other words, it should be declared as public void run(). A thread can be created by invoking the start() method on the object of the Thread class (or its derived class). When the JVM schedules the thread, it will move the thread to a runnable state and then execute the run() method. (We’ll discuss thread states later in this chapter). When the run() method completes its execution and returns, the thread will terminate.

class MyThread1 extends Thread {

public void run() {

try {

sleep(1000);

}

catch (InterruptedException ex) {

ex.printStackTrace();

// ignore the InterruptedException - this is perhaps the one of the

// very few of the exceptions in Java which is acceptable to ignore

}

System.out.println("In run method; thread name is: " + getName());

}

public static void main(String args[]) {

Thread myThread = new MyThread1();

myThread.start();

System.out.println("In main method; thread name is: " +

Thread.currentThread().getName());

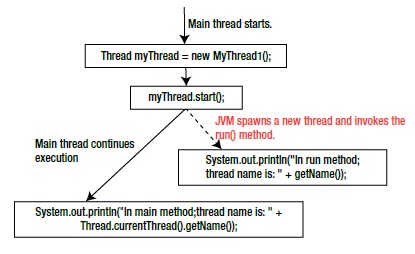
}

}

This program prints the following:

In main method; thread name is: main

In run method; thread name is: Thread-0



### Implementing the Runnable Interface

The Thread class itself implements the Runnable interface. Instead of extending the Thread class, you can implement the Runnable interface. The Runnable interface declares a sole method, run().

// in java.lang package

public interface Runnable {

public void run();

}

When you implement the Runnable interface, you need to define the run() method. Remember Runnable does not declare the start() method. So, how do you create a thread if you implement the Runnable interface? Thread has an overloaded constructor, which takes a Runnable object as an argument.

Thread(Runnable target)

You can use this overloaded constructor to create a thread from a class that implements the Runnable interface. First, let’s change the previous program by implementing the Runnable interface. If you change “class MyThread1 extends Thread” to “class MyThread1 implements Runnable” and compile the code, you get two compiler errors:

MyThread1.java:3: cannot find symbol

symbol : method getName()

location: class MyThread1

System.out.println("In run method; thread name is: " + this.getName());

MyThread1.java:7: incompatible types

found : MyThread1

required: java.lang.Thread

Thread myThread = new MyThread1();

The getName() method is available in the Thread class, but the MyThread1 class does not extend Thread anymore, so it results in a compiler error. Similarly, the start() method is available in the Thread class, and you don’t have that method any more since you directly implement Runnable.

class MyThread2 implements Runnable {

public void run() {

System.out.println("In run method; thread name is: " +

Thread.currentThread().getName());

}

public static void main(String args[]) throws Exception {

Thread myThread = new Thread(new MyThread2());

myThread.start();

System.out.println("In main method; thread name is: " +

Thread.currentThread().getName());

}

}

It prints the same output as the previous program:

In main method; thread name is: main

In run method; thread name is: Thread-0

You are implementing the run() method like the previous program. However, to get the name of the string, you must follow a round-about route and get the thread name with Thread.currentThread().getName(), as you did in the case of getting the thread name in the main() method. Similarly, in the main() method, to create a thread you must pass the object of the class to the Thread constructor. It was easy and convenient to just create the MyThread1 object and call the start() method on that while extending the Thread class.

### Should you extend the Thread or implement the Runnable?

You can either extend the Thread class or implement the Runnable interface to create a thread. So, which one do you choose? The Thread class has the default implementation of the run() method, so if you don’t provide a definition of the run() method while extending the Thread class, the compiler will not complain. However, the default implementation in the Thread class does nothing, so if you want your thread to do some meaningful work, you need to still define it. The Runnable interface declares the run() method, so you must define the run() method in your class if you implement the Runnable interface. So it doesn’t matter if you implement Runnable or extend Thread. You have to define the run() method for all practical reasons. In summary, that is not a major difference between extending a Thread and implementing Runnable. How about an inheritance relationship?

Since Java supports only single inheritance, if you extend from Thread, you cannot extend from any other class. Since inheritance is an is-a relationship, you will rarely need the class to have an is-a relationship with the Thread class. So OOP purists argue that you should not extend the Thread class. On the other hand, if you implement the Runnable interface, you can still extend some other class. So, many Java experts suggest that it is better to implement the Runnable interface unless there are some strong reasons to extend the Thread class. However, extending the Thread class is more convenient in many cases.

Both the techniques are useful and mostly equivalent for problem solving. So take a practical perspective here: use either of them as needed for the specific problem you are trying to solve

## The Start( ) and Run( ) Methods

You override the run() method but invoke the start() method. Why can’t you directly call the run() method? If you change the previous program by only changing myThread.start() to myThread.run(), what will happen?

class MyThread3 implements Runnable {

public void run() {

System.out.println("In run method; thread name is: " +

Thread.currentThread().getName());

}

public static void main(String args[]) throws Exception {

Thread myThread = new Thread(new MyThread3());

myThread.run(); // note run() instead of start() here

System.out.println("In main method; thread name is : " +

Thread.currentThread().getName());

}

}

This prints the following:

In run method; thread name is: main

In main method; thread name is: main

Now the output is different! If you call the run() method directly, it simply executes as part of the calling thread. It does not execute as a thread: it doesn’t get scheduled and get called by the JVM. That is why the getName() method in the run() method returns “main” instead of “Thread-0.” When you call the start() method, the thread gets scheduled and the run() method is invoked by the JVM when it is time to execute that thread.

Never call the run() method directly for invoking a thread. Use the start() method and leave it to the JVM to implicitly invoke the run() method. Calling the run() method directly instead of calling start() is a mistake and is fairly common bug.

## Thread Name, Priority, and Group

You need to understand three main aspects associated with each Java thread: its name, priority, and the thread group to which it belongs. Every thread has a name, which you can used to identify the thread. If you do not give a name explicitly, a thread will get a default name. The priority can vary from 1, the lowest, to 10, the highest. The priority of the normal thread is by default 5, and you can change this default priority value by explicitly providing a priority value. Every thread is part of a thread group. It’s a rarely used feature, so we won’t cover it in this book. The toString() method of Thread prints these three details.

class SimpleThread {

public static void main(String []s) {

Thread t = new Thread();

System.out.println(t);

}

}

This program prints the following:

Thread[Thread-0,5,main]

Thread is the name of the class. Within “[“ and ”]” is the name of the thread, its priority, and the thread group. You did not give any name to the thread, so the default name Thread-0 was given (as you create more threads, threads will be given names like Thread-1, Thread-2, etc). The default priority is 5. You created the thread in main(), so the default thread group is “main.”

Now let’s try changing the name and priority of the thread using the setName() and setPriority() methods:

Thread t = new Thread();

t.setName("SimpleThread");

t.setPriority(9);

System.out.println(t);

This code segment prints the following:

Thread[SimpleThread,9,main]

The thread has the name and priority that you gave it. You can change the name of the threads as you wish and it does not change the behavior of the program. However, you need to be careful in changing thread priority since it can affect scheduling of threads. You can programmatically access the minimum, normal, and maximum priority of the threads using the static members MIN\_PRIORITY, NORM\_PRIORITY, and MAX\_PRIORITY.

## The States of a Thread

A thread has various states during its lifetime. It is important to understand the different states of a thread and learn to write robust code based on that understanding. You’ll see three thread states—new, runnable and terminated—which are applicable to almost all threads you will create in this section. We will discuss more thread states later.

A program can access the state of the thread using Thread.State enumeration. The Thread class has the getState() instance method, which returns the current state of the thread.

class BasicThreadStates extends Thread {

public static void main(String []s) throws Exception {

Thread t = new Thread(new BasicThreadStates());

System.out.println("Just after creating thread; \n" +

" The thread state is: " + t.getState());

t.start();

System.out.println("Just after calling t.start(); \n" +

" The thread state is: " + t.getState());

t.join();

System.out.println("Just after main calling t.join(); \n" +

" The thread state is: " + t.getState());

}

}

This program prints the following:

Just after creating thread;

The thread state is: NEW

Just after calling t.start();

The thread state is: RUNNABLE

Just after main calling t.join();

The thread state is: TERMINATED

Just after the creation of the thread and just before calling the start() method on that thread, the thread is in the new state. After calling the start() method, the thread is ready to run or is in the running state (which you cannot determine), so it is in runnable state. From the main() method, you are calling t.join(). The main() method waits for the thread t to die. So, once the statement t.join() successfully gets executed by the main() thread, it means that the thread t has died or terminated. So, the thread is in the terminated state now.

A word of advice: be careful about accessing the thread states using the getState() method. Why? By the time you acquire information on a thread state and print it, the state could have changed! We know the last statement is confusing. To understand the problem with getting thread state information using the getState() method, consider the previous example. In one sample run of the same program, it printed the following:

Just after creating thread;

The thread state is: NEW

Just after calling t.start();

The thread state is: TERMINATED

Just after main calling t.join();

The thread state is: TERMINATED

The State enum from the Thread class has the following values :

* NEW
* RUNNABLE
* BLOCKED
* WAITING
* TIMED\_WAITING
* TERMINATED

# Thread Safety

## Data Races

Threads share memory, and they can concurrently modify data. Since the modification can be done at the same time without safeguards, this can lead to unintuitive results. When two or more threads are trying to access a variable and one of them wants to modify it, you get a problem known as a data race (also called as race condition or race hazard).

Example:

// This class exposes a publicly accessible counter

// to help demonstrate data race problem

class Counter {

public static long count = 0;

}

// This class implements Runnable interface

// Its run method increments the counter three times

class UseCounter implements Runnable {

public void increment() {

// increments the counter and prints the value

// of the counter shared between threads

Counter.count++;

System.out.print(Counter.count + " ");

}

public void run() {

increment();

increment();

increment();

}

}

// This class creates three threads

public class DataRace {

public static void main(String args[]) {

UseCounter c = new UseCounter();

Thread t1 = new Thread(c);

Thread t2 = new Thread(c);

Thread t3 = new Thread(c);

t1.start();

t2.start();

t3.start();

}

}

Note that the values will usually be different every time you run this program; when we ran it two times, we got these outputs:

3 3 5 6 3 4 7 8 9

3 3 3 6 7 5 8 4 9

## Thread Synchronization

Java has a keyword, synchronized, that helps in thread synchronization. You can use it in two forms—synchronized blocks and synchronized methods.

### Synchronized Blocks

In synchronized blocks, you use the synchronized keyword for a reference variable and follow it by a block of code. A thread has to acquire a lock on the synchronized variable to enter the block; when the execution of the block completes, the thread releases the lock. For example, you can acquire a lock on this reference if the block of code is within a non-static method:

synchronized(this) {

// code segment guarded by the mutex lock

}

What if an exception gets thrown inside the synchronized block? Will the lock get released? Yes, no matter whether the block is executed fully or an exception is thrown, the lock will be automatically released by the JVM. With synchronized blocks, you can acquire a lock on a reference variable only. If you use a primitive type, you will get a compiler error.

int i = 10;

synchronized(i) { /\* block of code here\*/}

For this code, you will get the following compiler error:

Lock.java:5: int is not a valid type's argument for the synchronized statement

found : int

required: reference

synchronized(i) { /\* block of code here\*/}

Here is an improved version of the program discussed in the previous section that performs synchronized access to Counter.count and does both read and write operations on that in a critical section. For that, you need to change only the increment method, as in

public void increment() {

// These two statements perform read and write operations

// on a variable that is commonly accessed by multiple threads.

// So, acquire a lock before processing this "critical section"

synchronized(this) {

Counter.count++;

System.out.print(Counter.count + " ");

}

}

Now the program prints the expected output correctly:

1 2 3 4 5 6 7 8 9

In the increment() method, you acquire a lock on the this reference before reading and writing to Counter.count. So, it is not possible for more than one thread to execute these statements at the same time. Since only one thread can acquire a lock and execute the “critical section” code block, the counter is incremented by only one thread at a given time; as a result, the program prints the values 1 to 9 correctly (without the data race problem).

### Synchronized Methods

An entire method can be declared synchronized. In that case, when the method declared as synchronized is called, a lock is obtained on the object on which the method is called, and it is released when the method returns to the caller.

Here is an example:

public synchronized void assign(int i) {

val = i;

}

Now the assign() method is a synchronized method. If you call the assign() method, it will acquire the lock on the this reference implicitly and then execute the statement val = i;. What happens if some other thread acquired the lock already? Just like synchronized blocks, if the thread cannot get the lock, it will be blocked and the thread will wait until the lock becomes available.

A synchronized method is equivalent to a synchronized block if you enclose the whole method body in a synchronized(this) block. So, the equivalent assign() method using synchronized blocks is

public void assign() {

synchronized(this) {

val = i;

}

}

Let’s get back to the Counter example. The increment() method can be rewritten as a synchronized method also:

// declaring the increment synchronized instead of using

// a synchronized statement for a block of code inside the method

public synchronized void increment() {

Counter.count++;

System.out.print(Counter.count + " ");

}

Now the program prints the expected output correctly:

1 2 3 4 5 6 7 8 9

### The Wait/Notify Mechanism

In multi-threaded programs, often there is a need for one thread to communicate to another thread. The wait/notify mechanism is useful when threads must communicate in order to provide a functionality.

The method wait() allows the calling thread to wait for the wait object (on which wait() is called). In other words, if you want to make a thread wait for another thread, you can ask it to wait for the wait object using the wait()method. A thread remains in the wait state until some another thread calls the notify() or notifyAll() method on the wait object.

Here’s an example :

// The CoffeeMachine class runs as an independent thread.

// Once the machine makes a coffee, it notifies the waiter to pick it up.

// When the waiter asks the coffee machine to make a coffee again,

// it starts all over again, and this process keeps goes on . . .

public class CoffeeMachine extends Thread {

static String coffeeMade = null;

static final Object lock = new Object();

private static int coffeeNumber = 1;

void makeCoffee() {

synchronized(CoffeeMachine.lock) {

if(coffeeMade != null) {

try {

System.out.println("Coffee machine: Waiting for waiter notification to deliver the coffee");

**CoffeeMachine.lock.wait();**

}

catch(InterruptedException ie) {

ie.printStackTrace();

}

}

coffeeMade = "Coffee No. " + coffeeNumber ++;

System.out.println("Coffee machine says: Made " + coffeeMade);

// once coffee is ready, notify the waiter to pick it up

**CoffeeMachine.lock.notifyAll();**

System.out.println("Coffee machine: Notifying waiter to pick the coffee ");

}

}

public void run() {

while(true) {

makeCoffee();

try {

System.out.println("Coffee machine: Making another coffee now");

// simulate the time taken to make a coffee by calling sleep method

Thread.sleep(10000);

}

catch(InterruptedException ie) {

// its okay to ignore this exception

// since we're not using thread interrupt mechanism

ie.printStackTrace();

}

}

}

}

// The Waiter runs as an independent thread

// It interacts with the CoffeeMachine to wait for a coffee

// and deliver the coffee once ready and request the coffee machine

// for the next one, and this activity keeps going on forever . . .

public class Waiter extends Thread {

public void getCoffee() {

synchronized(CoffeeMachine.lock) {

if(CoffeeMachine.coffeeMade == null) {

try {

// wait till the CoffeeMachine says (notifies) that

// coffee is ready

System.out.println("Waiter: Will get orders till coffee machine notifies me ");

**CoffeeMachine.lock.wait();**

}

catch(InterruptedException ie) {

// its okay to ignore this exception

// since we're not using thread interrupt mechanism

ie.printStackTrace();

}

}

System.out.println("Waiter: Delivering " + CoffeeMachine.coffeeMade);

CoffeeMachine.coffeeMade = null;

// ask (notify) the coffee machine to prepare the next coffee

**CoffeeMachine.lock.notifyAll();**

System.out.println("Waiter: Notifying coffee machine to make another one");

}

}

public void run() {

// keep going till the user presses ctrl-C and terminates the program

while(true) {

getCoffee();

}

}

}

## Deadlocks

Obtaining and using locks is tricky, and it can lead to lots of problems. One of the difficult (and common) problems is known as a deadlock. There are other problems such as livelocks and lock starvation, which we’ll briefly discuss in the next section.

A deadlock arises when locking threads result in a situation where they cannot proceed and thus wait indefinitely for others to terminate. Say, one thread acquires a lock on resource r1 and waits to acquire another on resource r2. At the same time, say there is another thread that has already acquired r2 and is waiting to obtain a lock on r1. Neither of the threads can proceed until the other one releases the lock, which never happens—so they are stuck in a deadlock.

## Livelocks

To help understand livelocks, let’s consider an analogy. Assume that there are two robotic cars that are programmed to automatically drive in the road. There is a situation where two robotic cars reach the two opposite ends of a narrow bridge. The bridge is so narrow that only one car can pass through at a time. The robotic cars are programmed such that they wait for the other car to pass through first. When both the cars attempt to enter the bridge at the same time, the following situation could happen: each car starts to enter the bridge, notices that the other car is attempting to do the same, and reverses! Note that the cars keep moving forward and backward and thus appear as if they’re doing lots of work, but there is no progress made by either of the cars. This situation is called a livelock.

Consider two threads t1 and t2. Assume that thread t1 makes a change and thread t2 undoes that change. When both the threads t1 and t2 work, it will appear as though lots of work is getting done, but no progress is made. This situation is called a livelock in threads.

The similarity between livelocks and deadlocks is that the process “hangs” and the program never terminates. However, in a deadlock, the threads are stuck in the same state waiting for other thread(s) to release a shared resource; in a livelock, the threads keep executing a task, and there is continuous change in the process states, but the application as a whole does not make progress.

## Lock Starvation

Consider the situation in which numerous threads have different priorities assigned to them (in the range of lowest priority, 1, to highest priority, 10, which is the range allowed for priority of threads in Java). When a mutex lock is available, the thread scheduler will give priority to the threads with high priority over low priority. If there are many high-priority threads that want to obtain the lock and also hold the lock for long time periods, when will the low-priority threads get a chance to obtain the lock? In other words, in a situation where low-priority threads “starve” for a long time trying to obtain the lock is known as lock starvation.

## Thread safety implementation approaches

### Re-entrancy

Writing code in such a way that it can be partially executed by a thread, reexecuted by the same thread or simultaneously executed by another thread and still correctly complete the original execution. This requires the saving of state information in variables local to each execution, usually on a stack, instead of in static or global variables or other non-local state. All non-local state must be accessed through atomic operations and the data-structures must also be reentrant.

Reentrancy means (in general) that do something, and then while you are still doing it, you do it again. In the case of locks it means you do something like this on a single thread:

Acquire a lock on "foo".

Do something

Acquire a lock on "foo". Note that we haven't released the lock that we previously acquired.

...

Release lock on "foo"

...

Release lock on "foo"

Here's a example in Java using primitive object locks / monitors ... which are reentrant:

Object lock = new Object();

...

synchronized (lock) {

...

doSomething(lock, ...)

...

}

public void doSomething(Object lock, ...) {

synchronized (lock) {

...

}

}

### Thread-local storage

Variables are localized so that each thread has its own private copy. These variables retain their values across subroutine and other code boundaries, and are thread-safe since they are local to each thread, even though the code which accesses them might be executed simultaneously by another thread.

### Mutual exclusion

Access to shared data is serialized using mechanisms that ensure only one thread reads or writes to the shared data at any time. Incorporation of mutal exclusion needs to be well thought out, since improper usage can lead to side-effects like deadlocks, livelocks and resource starvation.

### Atomic operations

Shared data are accessed by using atomic operations which cannot be interrupted by other threads. This usually requires using special machine language instructions, which might be available in a runtime library. Since the operations are atomic, the shared data are always kept in a valid state, no matter how other threads access it. Atomic operations form the basis of many thread locking mechanisms, and are used to implement mutual exclusion primitives.

### Immutable objects

The state of an object cannot be changed after construction. This implies both that only read-only data is shared and that inherent thread safety is attained. Mutable (non-const) operations can then be implemented in such a way that they create new objects instead of modifying existing ones. This approach is used by the string implementations in Java

# Serialization

In computer science, in the context of data storage, serialization is the process of translating data structures or object state into a format that can be stored (for example, in a file or memory buffer, or transmitted across a network connection link) and reconstructed later in the same or another computer environment. When the resulting series of bits is reread according to the serialization format, it can be used to create a semantically identical clone of the original object. For many complex objects, such as those that make extensive use of references, this process is not straightforward. Serialization of object-oriented objects does not include any of their associated methods with which they were previously inextricably linked.

This process of serializing an object is also called marshalling an object. The opposite operation, extracting a data structure from a series of bytes, is deserialization (which is also called unmarshalling).

The goals for serializing Java objects are to:

* Have a simple yet extensible mechanism.
* Maintain the Java object type and safety properties in the serialized form.
* Be extensible to support marshaling and unmarshaling as needed for remote objects.
* Be extensible to support simple persistence of Java objects.
* Require per class implementation only for customization.
* Allow the object to define its external format.

After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

A Serializable class must do the following:

* Implement the java.io.Serializable interface
* Identify the fields that should be serializable (Use the serialPersistentFields member to explicitly declare them serializable or use the transient keyword to denote fields.)
* Have access to the no-arg constructor of its first nonserializable superclass

The class can optionally define the following methods:

* A writeObject method to control what information is saved or to append additional information to the stream
* A readObject method either to read the information written by the corresponding writeObject method or to update the state of the object after it has been restored
* A writeReplace method to allow a class to nominate a replacement object to be written to the stream
* A readResolve method to allow a class to designate a replacement object for the object just read from the stream

The class of an Externalizable object must do the following:

* Implement the java.io.Externalizable interface
* Implement a writeExternal method to save the state of the object
* Implement a readExternal method to read the data written by the writeExternal method from the stream and restore the state of the object
* Have the writeExternal and readExternal methods be solely responsible for the format, if an externally defined format is written
* Have a public no-arg constructor
* A writeReplace method to allow a class to nominate a replacement object to be written to the stream
* A readResolve method to allow a class to designate a replacement object for the object just read from the stream

To be stored in an Object Stream, each object must implement either the Serializable or the Externalizable interface:

* For a Serializable class, Object Serialization can automatically save and restore fields of each class of an object and automatically handle classes that evolve by adding fields or supertypes. A serializable class can declare which of its fields are saved or restored, and write and read optional values and objects.
* For an Externalizable class, Object Serialization delegates to the class complete control over its external format and how the state of the supertype(s) is saved and restored.

The serializable fields of a class can be defined two different ways. Default serializable fields of a class are defined to be the non-transient and non-static fields.

Writing objects and primitives to a stream is a straightforward process. For example:

// Serialize today's date to a file.

FileOutputStream f = new FileOutputStream("tmp");

ObjectOutput s = new ObjectOutputStream(f);

s.writeObject("Today");

s.writeObject(new Date());

s.flush();

First an OutputStream, in this case a FileOutputStream, is needed to receive the bytes. Then an ObjectOutputStream is created that writes to the FileOutputStream. Next, the string "Today" and a Date object are written to the stream. More generally, objects are written with the writeObject method and primitives are written to the stream with the methods of DataOutput.

Special handling is required for arrays, enum constants, and objects of type Class, ObjectStreamClass, and String. Other objects must implement either the Serializable or the Externalizable interface to be saved in or restored from a stream.

Primitive data types are written to the stream with the methods in the DataOutput interface, such as writeInt, writeFloat, or writeUTF. Individual bytes and arrays of bytes are written with the methods of OutputStream. Except for serializable fields, primitive data is written to the stream in block-data records, with each record prefixed by a marker and an indication of the number of bytes in the record.

Reading an object from a stream, like writing, is straightforward:

// Deserialize a string and date from a file.

FileInputStream in = new FileInputStream("tmp");

ObjectInputStream s = new ObjectInputStream(in);

String today = (String)s.readObject();

Date date = (Date)s.readObject();

First an InputStream, in this case a FileInputStream, is needed as the source stream. Then an ObjectInputStream is created that reads from the InputStream. Next, the string "Today" and a Date object are read from the stream. Generally, objects are read with the readObject method and primitives are read from the stream with the methods of DataInput.

Primitive data types are read from the stream with the methods in the DataInput interface, such as readInt, readFloat, or readUTF. Individual bytes and arrays of bytes are read with the methods of InputStream. Except for serializable fields, primitive data is read from block-data records.

Serialization Example :