**\* What is Concurrency?**

Concurrency is the ability of a computer program to execute multiple tasks simultaneously. Each task is executed by a separate ***thread*** hence the terms ***multi-threaded*** program or ***multi-tasking*** program. Most computer programs are capable of multi-tasking. Here are some examples:

* A web browser (Chrome, IE, Firefox…) can load multiple websites at the same time. It allows you to scroll through a web page while playing a video in another page, and downloading some files in the background.
* A screen recorder program (e.g. CamStudio) is capturing screenshots (for generating video) while recording your voice from the microphone.
* A music player is playing a song while rendering compelling visualization effects on screen.
* An IDE (NetBeans, Eclipse…) allows you to write code while it is checking the syntax and compiling the project.

Multi-threaded programs are also called concurrent programs. Developing concurrent programs is more complex and requires more efforts than creating single-threaded ones.

And concurrent programming contains a set of techniques that helps programmers build concurrent programs easily with less effort.

**\* How Concurrency Works?**

At the operating system level, a program is a ***process*** or a group of processes. A process contains one or more threads. Each process has its own memory space and resources which are shared among threads.

Concurrency can be achieved by sharing CPU’s process time among threads using a mechanism called time slicing or interleaving. That means the CPU executes thread #1 a little bit, then execute thread #2 a little bit, then execute thread #3 a little bit, … then it returns to execute thread #1, and thread #2,… and so on. The order of execution among threads is undetermined, but it’s certainly that all threads will have CPU time. Therefore, the execution of threads is not sequential, but interleaved. And because the CPU runs extremely fast (billions of operations in a second) so it’s likely that all threads are running simultaneously.

With the time slicing mechanism, multi-threading is possible even on CPUs with single core.

**\* Advantages and Challenges of Concurrency:**

Concurrency makes software programs more efficient and robust. Here are the main advantages of concurrency:

* Increase performance and productivity as a concurrent program can complete more tasks in the same time as compared to single-threaded program.
* Allow concurrent programs to have high responsiveness, especially for those needs to deal with heavy input/output and user’s interactions on graphical user interface (GUI).
* Lead to more appropriate program structure in concurrent programming, as many problems can be solved by using concurrent threads.

However, concurrency also faces the following challenges:

* How to ensure data consistency as multiple threads can read and write shared data concurrently.
* How to avoid deadlock, livelock and starvation.

Concurrent programming needs to deal with these challenges properly so developing multi-threaded programs is not easy, but it’s certainly worth it.

**\* How Java Supports Concurrent Programming?**

The Java programming language supports for concurrent programming from the ground up. The Java Virtual Machine (JVM) is a single process running on the host operating system, and all Java programs are always started by one thread - the main thread. From this thread you can create other threads for your program. Java also allows you to create new processes.

The **Thread**class represents a unit of execution and the **Runnable** interface represents a task that can be executed by a thread. The **Object** class implements the wait() method that causes the current thread to wait until another thread invokes notify() or notifyAll() methods on an object.

The **synchronized** keyword can be used to safeguard a variable or a code block which is accessed by multiple threads in order to prevent thread interference and memory consistency errors. It is called ***synchronization***.

So Thread, Runnable, Object and synchronized are the fundamental elements of the low-level multi-threading API which is adequate for every basic tasks. And based on these elements, a high-level API is built for more advanced tasks. This API is implemented in the java.util.concurrent package, hence the ***Concurrency API*.**

Java also provides concurrent data structures in the Java Collections framework such as BlockingQueue, ConcurrentMap, etc.

And the fork/join framework is designed to help you take advantages of multiple processors by breaking a work into smaller pieces recursively.

**\* How to create a thread:**

There are two ways for creating a thread in Java: by extending the Thread class; and by implementing the Runnable interface. Both are in the java.lang package so you don’t have to use import statement.

Then you put the code that needs to be executed in a separate thread inside the run() method which is overridden from the Thread/Runnable. And invoke the start() method on a Thread object to put the thread into running status (alive).

The following class, ThreadExample1, demonstrates the first way:

public class ThreadExample1 extends Thread {

      public void run() {

            System.out.println("My name is: " + getName());

      }

      public static void main(String[] args) {

            ThreadExample1 t1 = new ThreadExample1();

            t1.start();

            System.out.println("My name is: " + Thread.currentThread().getName());

      }

}

Let me explain to you how this code is working. You see that the ThreadExample1 class extends the Thread class and overrides the run() method. Inside the run() method, it simply prints a message that includes the name of the thread, which is returned from the getName() method of the Thread class.

And now let’s see the main() method that is invoked when the program starts. It creates an instance of the ThreadExample1 class and call its start() method to put the thread into running state. And the last line prints a message that includes the name of the main thread - as I told you before - every Java program is started from a thread called main. The static method currentThread() returns the Thread object associated with the current thread.

Run this program and you will see the output as follows:

My name is: Thread-0

My name is: main

You see, there are actually 2 threads:

- Thread-0: is the name of the thread we created.

- main: is the name of the main thread that starts the Java program.

The thread Thread-0 terminates as soon as its run() method runs to complete, and the thread main terminates after the main() method completes its execution.

One interesting point is that, if you run this program again for several times, you will see sometimes the thread Thread-0 runs first, sometimes the thread main runs first. This can be recognized by the order of thread names in the output changes randomly. That means there’s no guarantee of which thread runs first as they are both started concurrently. You should bear in mind this behavior with regard to multi-threading context.

Now, let’s see the second way that uses the Runnable interface. In the code below, the ThreadExample2 class implements the Runnable interface and override the run() method:

public class ThreadExample2 implements Runnable {

      public void run() {

            System.out.println("My name is: " + Thread.currentThread().getName());

      }

      public static void main(String[] args) {

            Runnable task = new ThreadExample2();

            Thread t2 = new Thread(task);

            t2.start();

            System.out.println("My name is: " + Thread.currentThread().getName());

      }

}

As you can see, there’s a small difference as compared to the previous program: An object of type Runnable (the ThreadExample2 class) is created and passed to the constructor of a Thread object (t2). The Runnable object can be viewed as a task which is separated from the thread that executes the task.

The two programs behave the same. So what are the pros and cons of these two ways of creating a thread?

Here’s the answer:

- Extending the Thread class can be used for simple cases. It cannot be used if your class needs to extend another class because Java doesn’t allow multiple inheritances of class.

- Implementing the Runnable interface is more flexible as Java allows a class can both extend another class and implement one or more interfaces.

And remember that the thread terminates after its run() method returns. It is put into dead state and cannot be able to start again. You can never restart a dead thread.

You can also set name for a thread either via constructor of the Thread class or via the setter method setName(). For example:

Thread t1 = new Thread("First Thread");

Thread t2 = new Thread();

t2.setName("Second Thread");

**\* How to pause a thread:**

You can make the currently running thread pauses its execution by invoking the static method sleep(milliseconds) of the Thread class. Then the current thread is put into sleeping state. Here’s how to pause the current thread:

try {

      Thread.sleep(2000);

} catch (InterruptedException ex) {

      // code to resume or terminate...

}

This code pauses the current thread for about 2 seconds (or 2000 milliseconds). After that amount of time, the thread returns to continue running normally.

InterruptedException is a checked exception so you must handle it. This exception is thrown when the thread is interrupted by another thread.

Let’s see a full example. The following NumberPrint program is updated to print 5 numbers, each after every 2 seconds:

public class NumberPrint implements Runnable {

      public void run() {

            for (int i = 1; i <= 5; i++) {

                  System.out.println(i);

                  try {

                        Thread.sleep(2000);

                  } catch (InterruptedException ex) {

                        System.out.println("I'm interrupted");

                  }

            }

      }

      public static void main(String[] args) {

            Runnable task = new NumberPrint();

            Thread thread = new Thread(task);

            thread.start();

      }

}

Note that you can’t pause a thread from another thread. Only the thread itself can pause its execution. And there’s no guarantee that the thread always sleep exactly for the specified time because it can be interrupted by another thread, which is described in the next section.

**\* How to interrupt a thread:**

Interrupting a thread can be used to stop or resume the execution of that thread from another thread. For example, the following statement interrupts the thread t1 from the current thread:

t1.interrupt();

If t1 is sleeping, then calling interrupt() on t1 will cause the InterruptedException to be thrown. And whether the thread should stop or resume depending on the handling code in the catch block.

In the following code example, the thread t1 prints a message after every 2 seconds, and the main thread interrupts t1 after 5 seconds:

public class ThreadInterruptExample implements Runnable {

      public void run() {

            for (int i = 1; i <= 10; i++) {

                  System.out.println("This is message #" + i);

                  try {

                        Thread.sleep(2000);

                        continue;

                  } catch (InterruptedException ex) {

                        System.out.println("I'm resumed");

                  }

            }

      }

      public static void main(String[] args) {

            Thread t1 = new Thread(new ThreadInterruptExample());

            t1.start();

            try {

                  Thread.sleep(5000);

                  t1.interrupt();

            } catch (InterruptedException ex) {

                  // do nothing

            }

      }

}

As you can see in the catch block in the run() method, it continues the for loop when the thread is interrupted:

try {

      Thread.sleep(2000);

} catch (InterruptedException ex) {

      System.out.println("I'm resumed");

      continue;

}

That means the thread resumes running while it is sleeping.

To stop the thread, just change the code in the catch block to return from the run() method like this:

try {

      Thread.sleep(2000);

} catch (InterruptedException ex) {

      System.out.println("I'm about to stop");

      return;

}

You see, the return statement causes the run() method to return which means the thread terminates and goes to dead state.

What if a thread doesn’t sleep (no handling for InterruptedException)?

In such case, you need to check the interrupt status of the current thread using either of the following methods of the Thread class:

- interrupted(): this static method returns true if the current thread has been interrupted, or false otherwise. Note that this method clears the interrupt status, meaning that if it returns true, then the interrupt status is set to false.

- isInterrupted(): this non-static method checks the interrupt status of the current thread and it doesn’t clear the interrupt status.

The ThreadInterruptExample above can be modified to use the checking method as below:

public class ThreadInterruptExample implements Runnable {

      public void run() {

            for (int i = 1; i <= 10; i++) {

                  System.out.println("This is message #" + i);

                  if (Thread.interrupted()) {

                        System.out.println("I'm about to stop");

                        return;

                  }

            }

      }

      public static void main(String[] args) {

            Thread t1 = new Thread(new ThreadInterruptExample());

            t1.start();

            try {

                  Thread.sleep(5000);

                  t1.interrupt();

            } catch (InterruptedException ex) {

                  // do nothing

            }

      }

}

However this version doesn’t behave the same as the previous one because the thread t1 terminates very quickly as it doesn’t sleep and the print statements are executed very fast. So this example is just to show you how it is used. In practice, this kind of checking on interrupt status should be applied for long-running operations such as IO, network, database, etc.

And remember that when the InterruptedException is thrown, the interrupt status is cleared.

If you look at the Thread class in Javadocs, you will see there are 4 methods:

destroy() - stop() - suspend() - resume()

However all these methods are deprecated, meaning that you shouldn’t use them. Let use the interruption mechanism I have described so far.

**\* How to make a thread waits other threads?**

This is called joining and is useful in case you want the current thread to wait for other threads to complete. After that the current thread continues running. For example:

                t1.join();

This statement causes the current thread to wait for the thread t1 to complete before it continues. In the following program, the current thread (main) waits for the thread t1 to complete:

public class ThreadJoinExample implements Runnable {

      public void run() {

            for (int i = 1; i <= 10; i++) {

                  System.out.println("This is message #" + i);

                  try {

                        Thread.sleep(2000);

                  } catch (InterruptedException ex) {

                        System.out.println("I'm about to stop");

                        return;

                  }

            }

      }

      public static void main(String[] args) {

            Thread t1 = new Thread(new ThreadJoinExample());

            t1.start();

            try {

                  t1.join();

            } catch (InterruptedException ex) {

                  // do nothing

            }

            System.out.println("I'm " + Thread.currentThread().getName());

      }

}

In this program, the current thread (main) always terminates after the thread t1 completes. Hence you see the message “I’m main” is always printed last:

This is message #1

This is message #2

This is message #3

This is message #4

This is message #5

This is message #6

This is message #7

This is message #8

This is message #9

This is message #10

I'm main

Note that the join() method throws InterruptedException if the current thread is interrupted, so you need to catch it.

There are 2 overloads of join() method:

- join(milliseconds)

- join(milliseconds,  nanoseconds)

These methods cause the current thread to wait at most for the specified time. That means if the time expires and the joined thread has not completed, the current thread continues running normally.

You can also join multiple threads with the current thread, for example:

t1.join();

t2.join();

t3.join();

In this case, the current thread has to wait for all three threads t1, t2 and t3 completes before it can resume running.

 Understanding Synchronization - The Problems of Unsynchronized Code

In a multi-threaded application, several threads can access the same data concurrently, which may leave the data in inconsistent state (corrupted or inaccurate). Let’s find out how multi-thread access can be a source of problems by going through an example that demonstrates the processing of transactions in a bank.

We have a class that represents an account in the bank as follows:

public class Account {

      private int balance = 0;

      public Account(int balance) {

            this.balance = balance;

      }

      public void withdraw(int amount) {

            this.balance -= amount;

      }

      public void deposit(int amount) {

            this.balance += amount;

      }

      public int getBalance() {

            return this.balance;

      }

}

The balance of an account can be changed frequently due to the transactions of deposit and withdrawal.

The following code represents a bank that manages some accounts:

public class Bank {

      public static final int MAX\_ACCOUNT = 10;

      public static final int MAX\_AMOUNT = 10;

      public static final int INITIAL\_BALANCE = 100;

      private Account[] accounts = new Account[MAX\_ACCOUNT];

      public Bank() {

            for (int i = 0; i < accounts.length; i++) {

                  accounts[i] = new Account(INITIAL\_BALANCE);

            }

      }

      public void transfer(int from, int to, int amount) {

            if (amount <= accounts[from].getBalance()) {

                  accounts[from].withdraw(amount);

                  accounts[to].deposit(amount);

                  String message = "%s transfered %d from %s to %s. Total balance: %d\n";

                  String threadName = Thread.currentThread().getName();

                  System.out.printf(message, threadName, amount, from, to, getTotalBalance());

            }

      }

      public int getTotalBalance() {

            int total = 0;

            for (int i = 0; i < accounts.length; i++) {

                  total += accounts[i].getBalance();

            }

            return total;

      }

}

As you can see, this bank consists of 10 accounts for each is initialized with a balance amount of 100. So the total balance of these 10 accounts is 10 x 100 = 1000.

The transfer() method withdraws a specified amount from an account and deposit that amount to the target account. The transfer will be processed if and only if the source account has enough balance. And after the transfer has been done, a log message is printed to let us know the transaction details.

The getTotalBalance() method returns the total amount of all accounts, which must be always 1000. We check this number after every transaction to make sure that the program runs correctly.

As the bank allows many transactions to happen at the same time, the following class represents a transaction:

public class Transaction implements Runnable {

      private Bank bank;

      private int fromAccount;

      public Transaction(Bank bank, int fromAccount) {

            this.bank = bank;

            this.fromAccount = fromAccount;

      }

      public void run() {

            while (true) {

                  int toAccount = (int) (Math.random() \* Bank.MAX\_ACCOUNT);

                  if (toAccount == fromAccount) continue;

                  int amount = (int) (Math.random() \* Bank.MAX\_AMOUNT);

                  if (amount == 0) continue;

                  bank.transfer(fromAccount, toAccount, amount);

                  try {

                        Thread.sleep(50);

                  } catch (InterruptedException e) {

                        e.printStackTrace();

                  }

            }

      }

}

As you can see, this Transaction class implements the Runnable interface so the code in its run() method can be executed by a separate thread.

The source account is passed from the constructor and the target account is chosen randomly, and both source and target cannot be the same. Also the amount to be transferred is chosen randomly but always less than 10. After the transaction has been done, the current thread goes to sleep for a very short time (50 milliseconds), and then it continues doing the same steps repeatedly until the thread is terminated.

And here’s the test program:

public class TransactionTest {

      public static void main(String[] args) {

            Bank bank = new Bank();

            for (int i = 0; i < Bank.MAX\_ACCOUNT; i++) {

                  Thread t = new Thread(new Transaction(bank, i));

                  t.start();

            }

      }

}

As you can see, a Bank instance is created and shared among the threads that perform the transactions. For each account, a new thread is created to transfer money from that account to other randomly chosen accounts. That means there are total 10 threads sharing one instance of Bank class. These threads will run forever until the program is terminated by pressing Ctrl + C.

**Remember this rule:** No matter how many transactions are processed, the total balance of all accounts must remain unchanged. In other words, the program must consistently report this number as 1000.

Now, let’s compile and run the TransactionTest program and observe the output. Initially you should see some output like this:

A screenshot of a computer

Description automatically generated

The total balance is reported as 1000 consistently.

But wait! Let the program continues running longer, you quickly see a problem happens:

A screenshot of a computer

Description automatically generated

Ouch! Somehow the total balance is getting changed. It doesn’t remain at 1000 anymore. It’s getting smaller and smaller over time. Why did this happen?

There must be something wrong with the program. Let’s analyze the code to find out why.

Look at the Transaction class, you see multiple threads execute the transfer() method of the shared instance of the Bank class:

bank.transfer(fromAccount, toAccount, amount);

This method is implemented as follows:

public void transfer(int from, int to, int amount) {

      if (amount <= accounts[from].getBalance()) {

            accounts[from].withdraw(amount);

            accounts[to].deposit(amount);

            // code to print the log message…

      }

}

Suppose that the account #1 has balance of 5 after some transactions. The thread #1 is executing the if statement to verify that account has sufficient fund to transfer and amount of 3. Since the account’s balance is 5, the thread #1 enters the body of the if block.

But just before the thread #1 executes the statement to withdraw:

accounts[from].withdraw(amount);

another thread (say thread #2) has performed a transaction that withdraws an amount of 4 from the account #1. Now the thread #1 executes the withdraw operation and at this time the balance is 5 - 4 = 1, which is no longer seen as 5 by the thread #1. Hence the balance of account #1 is now 1 - 3 = -2. The balance is negative, so that’s why when the program calculates the total balance again, it gets decreased!

If you keep running the program longer and longer, you will see the total balance can get smaller and smaller:

A screenshot of a computer

Description automatically generated

That means the shared data may get corrupted when it is updated by multiple threads concurrently.

A similar problem can happen with the deposit operation. Suppose that the thread #3 is about to add an amount of 8 to the account #3. Before adding, the thread #3 sees the balance of this account is 10. But just before the thread #3 updates the balance, another thread (say thread #4) performs a withdrawal of an amount of 5 on this account, so its balance is 10 - 5 = 5.

In the mean time, the thread #3 still sees the balance is 10 so it adds 8 to 10 which results the balance of the account #3 is 18. But an amount of 5 has been added to another account, which means the total balance gets increased by 5. That’s why you may also see that the total balance gets increased over time when the program keeps running.

Let run the test program several times and observe the output yourself. The output is unpredicted: sometimes you see the total balance gets increased, sometimes it gets decreased, and sometimes it goes up and goes down, whatever!

Also try to change the sleep time in the Transaction class. The longer time, the total balance gets changed slower. And the shorter time, the total balance gets changed faster.

So what should we do to fix this problem?

We need a mechanism that is able to guarantee that code in the transfer() method is executed by only one thread at a time. In other words, we need to synchronize access to shared data.

Using Lock and Condition Objects

So you see the problem with the bank transaction example described in the previous email. We need to protect the shared data which may get corrupted due to concurrent updates by multiple threads. Now, let’s see what solution Java provides to serialize access to the transfer() method of the Bank class.

**\* Using Lock with ReentrantLock Object**

The Java Concurrency API provides a synchronization mechanism that involves in locking/unlocking on a lock object like this:

class Clazz {

      private Lock lock = new ReentrantLock();

      public void method() {

            lock.lock();      // 1

            try {

                  // 2: code needs to be protected

            } finally {

                  lock.unlock();    // 3

            }

      }

}

Let me explain how this mechanism works. When a thread enters line 1, it attempts to acquire the lock object and if the lock is not held by another thread, the current thread gets exclusive ownership on the lock object. If the lock is currently held by another thread, then the current thread blocks and waits until the lock is released.

Once the current thread successfully acquires the lock, it executes the code in the try block without worrying about intervention of other threads. Finally the thread releases the lock and exits the method (line 3). After that, chance to acquire the lock is given to other threads. At any time, only one thread owns the lock and can execute the protected code. Other threads block and wait until the lock becomes available.

The unlock statement is placed inside the finally block in order to ensure that the thread eventually relinquishes the lock in case of an exception thrown.

Hence we update the Bank class as shown below:

import java.util.concurrent.locks.\*;

public class Bank {

      public static final int MAX\_ACCOUNT = 10;

      public static final int MAX\_AMOUNT = 10;

      public static final int INITIAL\_BALANCE = 100;

      private Account[] accounts = new Account[MAX\_ACCOUNT];

      private Lock bankLock;

      public Bank() {

            for (int i = 0; i < accounts.length; i++) {

                  accounts[i] = new Account(INITIAL\_BALANCE);

            }

            bankLock = new ReentrantLock();

      }

      public void transfer(int from, int to, int amount) {

            bankLock.lock();

            try {

                  if (amount <= accounts[from].getBalance()) {

                        accounts[from].withdraw(amount);

                        accounts[to].deposit(amount);

                        String message = "%s transfered %d from %s to %s. Total balance: %d\n";

                        String threadName = Thread.currentThread().getName();

                        System.out.printf(message, threadName, amount, from, to, getTotalBalance());

                  }

            } finally {

                  bankLock.unlock();

            }

      }

      public int getTotalBalance() {

            bankLock.lock();

            try {

                  int total = 0;

                  for (int i = 0; i < accounts.length; i++) {

                        total += accounts[i].getBalance();

                  }

                  return total;

            } finally {

                  bankLock.unlock();

            }

      }

}

Here, a ReentrantLock object is created as an instance variable of the class. The ReentrantLock class is an implementation of the Lock interface. Both are defined in the java.util.concurrent.locks package.

Look closer at the transfer() method which is safeguarded for concurrent access by using a lock object as follows:

public void transfer(int from, int to, int amount) {

      bankLock.lock();

      try {

            if (amount <= accounts[from].getBalance()) {

                  accounts[from].withdraw(amount);

                  accounts[to].deposit(amount);

                  String message = "%s transfered %d from %s to %s. Total balance: %d\n";

                  String threadName = Thread.currentThread().getName();

                  System.out.printf(message, threadName, amount, from, to, getTotalBalance());

            }

      } finally {

            bankLock.unlock();

      }

}

You can notice that this method calls the getTotalBalance() method which is also protected by the lock/unlock mechanism on the same bankLock object:

public int getTotalBalance() {

      bankLock.lock();

      try {

            int total = 0;

            for (int i = 0; i < accounts.length; i++) {

                  total += accounts[i].getBalance();

            }

            return total;

      } finally {

            bankLock.unlock();

      }

}

We also need to serialize access to the getTotalBalance() method in order to avoid a situation in which other threads reading the total balance while the current thread is updating an account’s balance which affects the total balance. In other words, no thread can access the getTotalBalance() method when the current threading is executing the transfer() method because both methods are locked by the same lock object bankLock.  
  
Now, let’s recompile the Bank class and then run the TransactionTest program, you will see that the problem of corrupted total balance has gone:

A screenshot of a computer

Description automatically generated

With the synchronization solution using lock object we have applied, the program is now running as expected: the total balance remains unchanged all the time.

**Why ReentrantLock?**

You may feel the name ReentrantLock a little bit difficult to understand, but it has a good reason for that name. The ReentrantLock allows a thread to acquire a lock it already owns multiple times recursively. Look at the transfer() method, you see that it calls the getTotalBalance() method, right? By entering the getTotalBalance() method, the current thread acquires the lock object two times, right?

The number of times that a thread acquires a lock is stored in a ***holdcount*** variable. When the thread acquires the lock, the hold count is increased by 1, and when it releases the lock, hold count is decreased by 1. The lock is completely relinquished if hold count is 0. So there must be a call to unlock() for every call to lock().

In the Bank class above, when the current thread acquires the lock in the transfer() method, hold count is 1; and when it acquires the lock in the getTotalBalance() method, hold count is 2. When the thread releases the lock in the getTotalBalance() method, hold count is 1. And when the thread releases the lock in the transfer() method, hold count is 0.

That’s why this lock is called reentrant.

**\* Locking with Condition object**

In our bank example, a transaction will be processed if and only if the account has enough balance to cover the transfer:

if (amount <= accounts[from].getBalance()) {

      // transfer money...

}

In case the account doesn’t have enough fund, what if we want the current thread to wait until other threads have made deposit to this account? This logic can be explained by the following pseudo-code:

lock.lock();

try {

      if (not sufficient fund) {

            // wait...

      }

      // transfer...

} finally {

      lock.unlock();

}

The Java Concurrency API allows us to achieve this by providing a **Condition** object which can be obtained from the lock object like this:

Condition availableFund = bankLock.newCondition();

If the condition (enough fund to transfer) has not been met, we can tell the current thread to wait by invoking this statement:

availableFund.await();

This causes the current thread blocks and waits, which means the current thread gives up the lock so other threads have chance to update the balance of this account. The current thread blocks until another thread calls:

availableFund.signal();

or:

availableFund.signalAll();

The difference between signal() and signalAll() is that the signal() method wakes up only one thread among the waiting ones. Which thread is chosen depends on the thread scheduler, which means there’s no guarantee that the current thread will wake up if one thread invokes signal().

And the signalAll() method wakes up all threads which are currently waiting. Note that it’s up to the thread scheduler decides which thread takes the turn. All threads awake but only one is granted access to the lock. That also means there’s no guarantee that the current thread can acquire the lock again though it is waken up. If it is the case, the thread continues blocking and waiting for other chances, until it gets the locks and exits the method.

Now, let’s update the Bank class as follows:

import java.util.concurrent.locks.\*;

public class Bank {

      public static final int MAX\_ACCOUNT = 10;

      public static final int MAX\_AMOUNT = 10;

      public static final int INITIAL\_BALANCE = 100;

      private Account[] accounts = new Account[MAX\_ACCOUNT];

      private Lock bankLock;

      private Condition availableFund;

      public Bank() {

            for (int i = 0; i < accounts.length; i++) {

                  accounts[i] = new Account(INITIAL\_BALANCE);

            }

            bankLock = new ReentrantLock();

            availableFund = bankLock.newCondition();

      }

      public void transfer(int from, int to, int amount) {

            bankLock.lock();

            try {

                  while (accounts[from].getBalance() < amount) {

                        availableFund.await();

                  }

                  accounts[from].withdraw(amount);

                  accounts[to].deposit(amount);

                  String message = "%s transfered %d from %s to %s. Total balance: %d\n";

                  String threadName = Thread.currentThread().getName();

                  System.out.printf(message, threadName, amount, from, to, getTotalBalance());

                  availableFund.signalAll();

            } catch (InterruptedException e) {

                  e.printStackTrace();

            } finally {

                  bankLock.unlock();

            }

      }

      public int getTotalBalance() {

            bankLock.lock();

            try {

                  int total = 0;

                  for (int i = 0; i < accounts.length; i++) {

                        total += accounts[i].getBalance();

                  }

                  return total;

            } finally {

                  bankLock.unlock();

            }

      }

}

Recompile this class and then run the TransactionTest program again and observe the result yourself.

So using Condition object would be useful in case you want the current thread to wait until the condition is met, rather than giving up immediately if it is not.

The technique we have experienced so far is called ***explicit locking mechanism***, which uses a concrete Lock object with a Condition object.

Using synchronized keyword

So you know how to use Lock and Condition objects for synchronizing access to a method. This is basically how synchronized is designed and working in Java. And to make it easier for programmers, Java provides the ***synchronized*** keyword that operates on the default lock of a class. This default lock is called ***intrinsic lock*** which belongs to every Java object.

The synchronized keyword can be used at method level or code block level. Let’s look at the first approach first.

**\* Synchronized Methods:**

Consider the following class:

public class A {

      public **synchronized** void update() {

            // code needs to be serialized for access

      }

}

Here, the update() method is synchronized. It is equivalent to the following code that uses a lock object explicitly:

public class A {

      public void update() {

            this.intrinsicLock.lock();

            try {

                  // code needs to be serialized for access

            } finally {

                  this.intrinsicLock.unlock();

            }

      }

}

Here, the intrinsic lock belongs to an instance of the class. And the following code explains how to use condition with a synchronized method:

public class A {

      public synchronized void update() {

            if (!condition) {

                  this.wait();

            }

            // code needs to be serialized for access

            this.notify();

            // or:

            this.notifyAll();

      }

}

The methods wait(), notify() and notifyAll() behaves in the same manner as the methods await(), signal() and signalAll() of a Lock object. These methods are provided by the Object class. So every object has its own intrinsic lock and intrinsic condition.

Now, the Bank class can be rewritten using the synchronized keyword as follows:

public class Bank {

      public static final int MAX\_ACCOUNT = 10;

      public static final int MAX\_AMOUNT = 10;

      public static final int INITIAL\_BALANCE = 100;

      private Account[] accounts = new Account[MAX\_ACCOUNT];

      public Bank() {

            for (int i = 0; i < accounts.length; i++) {

                  accounts[i] = new Account(INITIAL\_BALANCE);

            }

      }

      public synchronized void transfer(int from, int to, int amount) {

            try {

                  while (accounts[from].getBalance() < amount) {

                        wait();

                  }

                  accounts[from].withdraw(amount);

                  accounts[to].deposit(amount);

                  String message = "%s transfered %d from %s to %s. Total balance: %d\n";

                  String threadName = Thread.currentThread().getName();

                  System.out.printf(message, threadName, amount, from, to, getTotalBalance());

                  notifyAll();

            } catch (InterruptedException e) {

                  e.printStackTrace();

            }

      }

      public synchronized int getTotalBalance() {

            int total = 0;

            for (int i = 0; i < accounts.length; i++) {

                  total += accounts[i].getBalance();

            }

            return total;

      }

}

You see, using the synchronized keyword make the code more compact, right? But you wouldn’t understand how a synchronized method works without understanding about the explicit locking mechanism, would you?

Let recompile the Bank class and then run the TransactionTest program again, you will see that the program behaves the same way as the previous version which uses explicit locking mechanism. But you write much less code. It’s cool, isn’t it?

The following are some noteworthy points with regards to synchronized instance methods (non-static ones):

- When a thread is entering a synchronized method, it tries to acquire the intrinsic lock associate with the current instance of the class. If the thread successfully owns the lock, other threads will block when attempting to execute any synchronized instance methods of the class. That means if a class contains multiple synchronized instance methods, only one can be executed by a thread at a time.

- A thread must own the lock before calling wait(), notify() or notifyAll(). Otherwise an IllegalMonitorStateException is thrown.

- The wait() method cause the current thread to wait until it is woken up by a thread that calls notify() or notifyAll(). And while waiting, the thread can be interrupted by another thread. Hence we have to handle the InterruptedException.

- The notify() method causes the current thread to give up the lock so a randomly selected waiting thread is given the lock. That means there’s no guarantee that the thread that calls wait() is selected. It’s up to the thread scheduler.

- The notifyAll() method causes the current thread to release the lock and wakes up all other threads that are currently waiting. All threads have the chance.

**\* Synchronized Blocks:**

In case you want to synchronize access at a smaller scope, i.e. a block of code rather than the whole method, you can use the synchronized keyword like this:

public void update() {

      synchronized (obj) {

            // code block

      }

}

A thread must hold the lock associated with the object obj before it can execute the code block. The obj can be any kind of object which you want to use it as a lock.

And use the synchronized block with condition as follows:

synchronized (obj) {

      if (!condition) {

            obj.wait();

      }

      // code block

      obj.notify();

      // or:

      obj.notifyAll();

}

By using synchronized blocks you have greater control over which part of the code should be serialized for access. For example, you can block concurrent access to the write() method while allow the read() method to be executed concurrently:

public class A {

      private Object lock = new Object();

      public void write() {

            synchronized (lock) {

                  // code to write

            }

      }

      public void read() {

            // code to read

      }

}

Here, you can see a pure Object is used as a lock. The write() method can be executed by only one thread at a time, whereas the read() method can be executed by multiple threads concurrently. This is possible because when a thread is executing the synchronized block, it doesn’t necessarily have to own the lock associated with the instance of the class. Instead, it holds the lock associated with the object protected by the synchronized block.

Note that synchronizing a code block on the current instance of the class is equivalent to synchronizing an instance method. That means the following code:

public void update() {

      synchronized (this) {

            // code block

      }

}

is equivalent to this:

public synchronized void update() {

// code block

}

Hence the Bank class can be rewritten using synchronized blocks on the this instance. It’s your exercise.

**\* Synchronized Static Methods:**

You can synchronize a static method and for that the threads have to acquire a different lock: the lock associated with the class itself (static), not an instance of the class (this).

That means if you write:

public class A {

      public static synchronized void update() {

            // code

      }

}

is equivalent to:

public class A {

      public static void update() {

            synchronized (A.class) {

                  // code

            }

      }

}

So when a thread is executing a synchronized static method, it also blocks access to all other synchronized static methods. The synchronized non-static methods are still executable by other threads. It’s because synchronized static methods and synchronized non-static methods work on different locks: class lock and instance lock.

In other words, a synchronized static method and a non-static synchronized method will not block each other. They can run at the same time.

That’s how the intrinsic (implicit) locking mechanism works in Java.

**\* Explicit Locking vs. Intrinsic Locking:**

So far I have explained to you the work of two synchronization mechanism in Java:

- Explicit locking using Lock and Condition objects.

- Intrinsic locking using the synchronized keyword.

Now the question is: when to use which? When to use Lock and when to use synchronized?

Here are some guidelines that help you make your decision:

- Consider using the synchronized keyword if you want to block concurrent access to instance methods (non-static synchronized methods) or static methods (static synchronized methods).

- Consider using explicit Lock and Condition objects if you want to have greater control over the synchronization process:

                + Use more than one Condition objects associate with a Lock.

+ Specify a timeout while a thread is waiting. This means the thread can wake up itself after a specified timeout expires.

- Remember that using synchronized keyword is easier and less error-prone then using explicit lock. Using explicit lock gives you more control but you have to put more effort.

Understanding Deadlock, Livelock and Starvation

So far you have grasped the concepts of synchronization in Java throughout the bank account transaction example. You are able to use the explicit locking mechanism and synchronized keyword to solve memory consistence problems that can happen when a shared resource is updated by multiple threads concurrently.

Synchronization is useful to protect data from corrupted state. However it may cause problems if not properly used. These problems are classified as deadlock, livelock and starvation.

In today’s lesson, I’m going to help you identify each type of problem so you can know to avoid them.

**\* Understanding Deadlock**

***Deadlock*** describes a situation where two more threads are blocked because of waiting for each other forever. When deadlock occurs, the program hangs forever and the only thing you can do is to kill the program.

Let’s get back to our bank account transaction example. Modify the maximum amount can be transferred from 10 to 200 in the Bank class as follows:

public static final int MAX\_AMOUNT = 200;

Look at the Transaction class you see the amount is chosen randomly by this statement:

int amount = (int) (Math.random() \* Bank.MAX\_AMOUNT);

Now, recompile the Bank and Transaction classes, and then run the TransactionTest program. Guess what will happen?

You will see that the program runs for a few transactions and hangs forever, as shown in the following screenshot:  
  
A screenshot of a computer

Description automatically generated

The program encounters a deadlock and cannot continue. Why can deadlock happen when we increase the maximum amount of money can be transferred among accounts?

Let’s analyze the code to understand why.

In the Bank class you will each account is initialized with an amount of 100. Now the maximum amount can be transferred is 200, so there will be some threads trying to transfer an amount which is greater than the account’s balance, for example:

      Thread 1 tries to transfer 150 from account 1 to account 2

      Thread 2 tries to transfer 170 from account 3 to account 1

Account 1 has only 100 in balance so thread 1 has to wait for other threads to deposit more funds to this account. Similarly, thread 2 also has to wait because account 3 doesn’t have sufficient fund. Other threads may add funds to accounts 1 and 3, but if all threads are trying to transfer an amount greater than the account’s balance, they are waiting for each other forever. Hence deadlock occurs.

That’s why you see the program quickly runs into deadlock after few transactions have been done. It hangs and you have to press Ctrl + C to terminate the program.

You can ask why the previous version of the example runs fine. It’s because the maximum account is smaller (10) than the balance (100), so all accounts have enough fund to transfer.

Another common reason for deadlock problem is two or more threads attempt to acquire two locks simultaneously, but in different order. Consider the following class:

public class Business {

      private Object lock1 = new Object();

      private Object lock2 = new Object();

      public void foo() {

            synchronized (lock1) {

                  synchronized (lock2) {

                        System.out.println("foo");

                  }

            }

      }

      public void bar() {

            synchronized (lock2) {

                  synchronized (lock1) {

                        System.out.println("bar");

                  }

            }

      }

}

As you can see, both the methods foo() and bar() try to acquire two lock objects lock1 and lock2 but in different order.

And consider the following test program:

public class BusinessTest1 {

      public static void main(String[] args) {

            Business business = new Business();

            Thread t1 = new Thread(new Runnable() {

                  public void run() {

                        business.foo();

                  }

            });

            t1.start();

            Thread t2 = new Thread(new Runnable() {

                  public void run() {

                        business.bar();

                  }

            });

            t2.start();

      }

}

This program creates two threads, one executes the foo() method and another executes the bar() method on a shared instance of the Business class. But deadlock is likely never to occur because one thread can execute and exit a method very quickly so the other thread have chance to acquire the locks.

Let’s modify this test program in order to create 10 threads for executing foo() and other 10 threads for executing bar() as follows:

public class BusinessTest2 {

      public static void main(String[] args) {

            Business business = new Business();

            for (int i = 0; i < 10; i++) {

                  new Thread(new Runnable() {

                        public void run() {

                              business.foo();

                        }

                  }).start();

            }

            for (int i = 0; i < 10; i++) {

                  new Thread(new Runnable() {

                        public void run() {

                              business.bar();

                        }

                  }).start();

            }

      }

}

Run this program several times (4-10 times), you will see that sometimes the program runs fine:  
  
A screenshot of a computer

Description automatically generated

But sometimes it hangs like this:  
  
A screenshot of a computer

Description automatically generated

Why? It’s because deadlock happens. Let me explain how:

- Thread 1 enters foo() method and it acquires lock1. At the same time, thread 2 enters bar() method and it acquires lock2.

- Thread 1 tries to acquire lock2 which is currently held by thread 2, hence thread 1 blocks.

- Thread 2 tries to acquire lock1 which is currently held by thread 1, hence thread 2 blocks.

Both threads block each other forever, deadlock occurs and the program hangs.

**So how to avoid deadlock?**

Java doesn’t have anything to escape deadlock state when it occurs, so you have to design your program to avoid deadlock situation. Avoid acquiring more than one lock at a time. If not, make sure that you acquire multiple locks in consistent order. In the above example, you can avoid deadlock by synchronize two locks in the same order in both methods:

public void foo() {

      synchronized (lock1) {

            synchronized (lock2) {

                  System.out.println("foo");

            }

      }

}

public void bar() {

      synchronized (lock1) {

            synchronized (lock2) {

                  System.out.println("bar");

            }

      }

}

Also try to shrink the synchronized blocks as small as possible to avoid unnecessary locking on code that doesn’t need to be synchronized.

**\* Understanding Livelock:**

***Livelock*** describes situation where two threads are busy responding to actions of each other. They keep repeating a particular code so the program is unable to make further progress:

Thread 1 acts as a response to action of thread 2

Thread 2 acts as a response to action of thread 1

Unlike deadlock, threads are not blocked when livelock occurs. They are simply too busy responding to each other to resume work. In other words, the program runs into an infinite loop and cannot proceed further.

Let’s see an example: a criminal kidnaps a hostage and he asks for ransom in order to release the hostage. A police agrees to give the criminal the money he wants once the hostage is released. The criminal releases the hostage only when he gets the money. Both are waiting for each other to act first, hence livelock.

Here’s the code of this example.

Criminal class:

public class Criminal {

      private boolean hostageReleased = false;

      public void releaseHostage(Police police) {

            while (!police.isMoneySent()) {

                  System.out.println("Criminal: waiting police to give ransom");

                  try {

                        Thread.sleep(1000);

                  } catch (InterruptedException ex) {

                        ex.printStackTrace();

                  }

            }

            System.out.println("Criminal: released hostage");

            this.hostageReleased = true;

      }

      public boolean isHostageReleased() {

            return this.hostageReleased;

      }

}

Police class:

public class Police {

      private boolean moneySent = false;

      public void giveRansom(Criminal criminal) {

            while (!criminal.isHostageReleased()) {

                  System.out.println("Police: waiting criminal to release hostage");

                  try {

                        Thread.sleep(1000);

                  } catch (InterruptedException ex) {

                        ex.printStackTrace();

                  }

            }

            System.out.println("Police: sent money");

            this.moneySent = true;

      }

      public boolean isMoneySent() {

            return this.moneySent;

      }

}

Test class:

public class HostageRescueLivelock {

      static final Police police = new Police();

      static final Criminal criminal = new Criminal();

      public static void main(String[] args) {

            Thread t1 = new Thread(new Runnable() {

                  public void run() {

                        police.giveRansom(criminal);

                  }

            });

            t1.start();

            Thread t2 = new Thread(new Runnable() {

                  public void run() {

                        criminal.releaseHostage(police);

                  }

            });

            t2.start();

      }

}

Run this program and you will see that it runs into a loop which never terminates:  
  
A screenshot of a computer

Description automatically generated

So how to avoid livelock? There’s no general guideline, you have to design your program to avoid livelock situation.

**\* Understanding Starvation**

***Starvation*** describes a situation where a greedy thread holds a resource for a long time so other threads are blocked forever. The blocked threads are waiting to acquire the resource but they never get a chance. Thus they starve to death.

Starvation can occur due to the following reasons:

- Threads are blocked infinitely because a thread takes long time to execute some synchronized code (e.g. heavy I/O operations or infinite loop).

- A thread doesn’t get CPU’s time for execution because it has low priority as compared to other threads which have higher priority.

- Threads are waiting on a resource forever but they remain waiting forever because other threads are constantly notified instead of the hungry ones.

When a starvation situation occurs, the program is still running but doesn’t run to completion because some threads are not executed.

Let’s see an example. Suppose we have a Worker class like this:

public class Worker {

      public synchronized void work() {

            String name = Thread.currentThread().getName();

            String fileName = name + ".txt";

            try (

                  BufferedWriter writer = new BufferedWriter(new FileWriter(fileName));

            ) {

                  writer.write("Thread " + name + " wrote this mesasge");

            } catch (IOException ex) {

                  ex.printStackTrace();

            }

            while (true) {

                  System.out.println(name + " is working");

            }

      }

}

This class has a synchronized method work() that creates a text file .txt and writes a message to it. Then it repeatedly prints a message:

is working

And the following program creates 10 threads that call the work() method on a shared instance of the Worker class:

public class StarvationExample {

      public static void main(String[] args) {

            Worker worker = new Worker();

            for (int i = 0; i < 10; i++) {

                  new Thread(new Runnable() {

                        public void run() {

[worker.work](http://worker.work/" \t "_blank)();

                        }

                  }).start();

            }

      }

}

Compile and run this program and you will see that there’s only one thread gets executed:  
  
A screen shot of a computer

Description automatically generated

According to the code logic, each thread should create a text file with the name of .txt but you see only one gets created, e.g. thread-1.txt. That means other threads are unable to execute the work() method.

Why does this happen? It’s because the while loop runs forever so that the first executed thread never release the lock, causing other threads to wait forever.

A solution to solve this starvation problem is to make the current thread waits for a specified amount of time so other threads have chance to acquire the lock on the Worker object:

while (true) {

      System.out.println(name + " is working");

      try {

            wait(1000);

      } catch (InterruptedException ex) {

            ex.printStackTrace();

      }

}

Recompile and run this program again and you will see that all threads get executed, proven by 10 text files created and in the output:  
  
A screen shot of a computer screen

Description automatically generated

In general, you should design your program to avoid starvation situation.

**\* Summary**

So far I have helped you identify the 3 problems which can happen in multi-threading Java programs: deadlock, livelock and starvation.  Livelock and starvation are less common than deadlock but they still can occur. To summarize, the following points help you understand the key differences of these problems:

- **Deadlock**:  All threads are blocked, the program hangs forever.

- **Livelock**: No threads blocked but they run into infinite loops. The program is still running but unable to make further progress.

- **Starvation**: Only one thread is running, and other threads are waiting forever.

You should be aware of these problems which can occur with multiple threads and synchronization, and design your programs to avoid them.

**\* More about thread creation:**

You know that there are two ways for creating a thread: by extending the Threadclass and by implementing the Runnable interface. Consider the following simple program:

public class SimpleThreadExample {

      public static void main(String[] args) {

            Thread t = new Thread(new Task());

            t.start();

      }

}

class Task implements Runnable {

      public void run() {

            System.out.println(Thread.currentThread().getName());

      }

}

This program simply creates and starts a new thread which prints its name when running. This is a standard way to create and start a thread: the runnable class is created separately.

In addition, the Java programming language provides a more flexible way that makes use of anonymous class when creating a new thread. For example, the previous program can be re-written like this:

public class SimpleThreadExample {

      public static void main(String[] args) {

            new Thread(new Runnable() {

                  public void run() {

                        System.out.println(Thread.currentThread().getName());

                  }

            }).start();

      }

}

Here, we create an anonymous class (no-name class) that implements the Runnable interface, in place of the Thread’s constructor. Then we call start() immediately right after the new Thread object. Use this “shortcut” style if you have very few code in the run() method.

Since Java 8 with functional interface and Lambda expression, you can write even more compact code like this:

public class SimpleThreadExample {

      public static void main(String[] args) {

            new Thread(() -> {

                        System.out.println(Thread.currentThread().getName());

            }).start();

      }

}

Because Runnable is a functional interface (has only method), so the compiler easily infers the actual implementation from this code:

new Thread(() -> {

      System.out.println(Thread.currentThread().getName());

}).start();

Remember don’t overuse this “shortcut” way if you have quite a lot of code in the run() method, which can make the code less readable.

**\* Thread States (or Thread Life Cycle):**

A thread can go through various states during its life. The Thread’s getState() method returns an enum constant that indicates current state of the thread, which falls in one of the following values:

- RUNNABLE

- BLOCKED

- WAITING

- TIMED\_WAITING

- TERMINATED

Let me explain each state in details.

- **NEW**: when a thread is created but has not executed (the start() method has not been invoked), it is in the new state.

- **RUNNABLE**: when the start() method has been invoked, the thread enters the runnable state, and its run() method is executing. Note that the thread can come back to runnable state from another state (waiting, blocked), but it may not be picked immediately by the thread scheduler, hence the term “runnable”, not running.

- **BLOCKED**: when a thread tries to acquire an intrinsic lock (not a lock in the java.util.concurrent package) that is currently held by another thread, it becomes blocked. When all other threads have relinquished the lock and the thread scheduler has allowed this thread to hold the lock, the thread becomes unblocked and enters the runnable state.

- **WAITING**: a thread enters this state if it waits to be notified by another thread, which is the result of calling Object.wait() or Thread.join(). The thread also enters waiting state if it waits for a Lock or Condition in the java.util.concurrent package. When another thread calls Object‘s notify()/notifyAll() or Condition’s signal()/signalAll(), the thread comes back to the runnable state.

- **TIMED\_WAITING**: a thread enters this state if a method with timeout parameter is called: sleep(), wait(), join(), Lock.tryLock() and Condition.await(). The thread exits this state if the timeout expires or the appropriate notification has been received.

- **TERMINATED**: a thread enters terminated state when it has completed execution. The thread terminates for one of two reasons:

                + the run() method exits normally.

                + the run() method exits abruptly due to a uncaught exception occurs.

The following diagram helps you visually understand the thread states and transitions between them:

A diagram of a runable process

Description automatically generated

And the following code example illustrates how to check state of a thread:

public class ThreadState {

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

            Thread t = new Thread(new Runnable() {

                  public void run() {

                        Thread self = Thread.currentThread();

                        System.out.println(self.getName() + "is " + self.getState());

                  }

            });

            System.out.println(Thread.currentThread().getName() + "is " + t.getState());

            t.start();

            t.join();

            if (t.getState() == Thread.State.TERMINATED) {

                  System.out.println(t.getName() + " is terminated");

            }

      }

}

Run this program and you will see the following output:

Thread-0 is NEW

Thread-0 is RUNNABLE

Thread-0 is terminated

Note that the thread’s state may change after the call to getState(). That means calling getState()  may not reflect the actual state of the thread only a moment later.

**\* Thread Priorities:**

Each thread has a priority value that hints the thread scheduler how much it should be cared in case of many threads are running.

When the thread scheduler needs to pick a thread to run next among the waiting ones, it prefers the thread with highest priority. By default, a new thread has same priority as the thread created it. For example, if you create a new thread from the main method, the new thread has priority equal to the main thread’s priority.

You can set priority of a thread by calling Thread.setPriority(int priorityLevel). The priorityLevel ranges from minimum value (Thread.MIN\_PRIORITY = 1) to maximum value (Thread.MAX\_PRIORITY = 10). The Thread class also defines a constant indicating normal priority (Thread.NORM\_PRIORITY = 5). That means you can set any value from 1 to 10.

For example, the following code sets priority of thread t1 to a level above normal and priority of thread t2 to maximum:

Thread t1 = newThread;

t1.setPriority(8);

Thread t2 = newThread;

t2.setPriority(Thread.MAX\_PRIORITY);

To check priority of a thread, call getPriority() method. For example, the following code gets the priority of the current thread:

int priority = Thread.currentThread().getPriority();

If the threads have same priority, it’s up to for thread schedule to pick which one first (randomly).

Be careful when using thread priorities, because starvation can occur when low-priority threads do not have chance to run because the high-priority threads take all the CPU time.

And don’t overuse thread priorities because they are mapped to thread priorities of the host operating system, and each operating system treats thread priorities differently. So never structure your program relying on thread priorities.

**\* Daemon Thread:**

Java defines two types of thread: user thread (normal thread) and daemon thread. By default, when you create a new thread it is user thread. The Java Virtual Machine (JVM) won’t terminate if there are still user threads running. But it will exit if there are only daemon threads running.

Daemon threads have lower priority than normal ones, so they are used for running background services that serve user threads. An example of daemon thread in the JVM is the garbage collector thread that runs silently in the background to free unused memory.

You can make a thread daemon by calling Thread.setDaemon(true) and check daemon status by using isDaemon(). Note that setDaemon() should be called before the thread is started. Here’s an example:

Thread daemonThread = new Thread(new Runnable() {

      public void run() {

            // do something

      }

});

daemonThread.setDaemon(true);

daemonThread.start();

A thread inherits daemon status from its parent thread - the one that created it. The purpose of daemon threads is serving user threads, there’s no need to keep daemon threads running if all user threads terminate. That’s why the JVM will exit if there are only daemon threads running.

For example, the following program helps you understand the concept of daemon thread:

public class DaemonThread {

      public static void main(String[] args) {

            Thread userThread = new Thread(new Runnable() {

                  public void run() {

                        int x = 10;

                        while (x > 0) {

                              System.out.println("User thread: " + x--);

                              try {

                                    Thread.sleep(1000);

                              } catch (InterruptedException ex) { ex.printStackTrace(); }

                        }

                  }

            });

            userThread.start();

            Thread daemonThread = new Thread(new Runnable() {

                  public void run() {

                        while (true) {

                              System.out.println("Daemon thread is running...");

                              try {

                                    Thread.sleep(100);

                              } catch (InterruptedException ex) { ex.printStackTrace(); }

                        }

                  }

            });

            daemonThread.setDaemon(true);

            daemonThread.start();

      }

}

This program creates and starts two threads: userThread and daemonThread.  Look at the run() method of the daemonThread class you see that it will run forever (indefinite while loop). And the run() method of the userThread class will run for only 10 seconds.

Run this program and you will see it will terminate after 10 seconds when the userThread terminates, despite that the daemonThread is still running. That proves the JVM exits when only daemon threads are running.

Understanding ThreadGroup

ThreadGroup is a convenient class that groups some related threads as a single unit and allows you to perform some operations on a group as a whole, rather than with each separate thread.

You need to specify the name of the group upon creation like this:

ThreadGroup groupA = new ThreadGroup("Group A");

ThreadGroup groupB = new ThreadGroup("Group B");

And when you create a new thread, specify the thread group to which the thread belongs using the following Thread constructors:

                Thread(ThreadGroup group, String name)

Thread(ThreadGroup group, Runnable target)

Thread(ThreadGroup group, Runnable target, String name)

For example, suppose Task is a thread, you can create 4 threads and group them in one group like this:

ThreadGroup group = new ThreadGroup("GroupA");

new Task(group, "A").start();

new Task(group, "B").start();

new Task(group, "C").start();

new Task(group, "D").start();

For more complex need, you can also create a tree of thread groups using the following ThreadGroup constructor:

ThreadGroup(ThreadGroup parent, String name)

For example, the following code creates a tree consisting of 2 groups:

ThreadGroup base = new ThreadGroup("Base");

ThreadGroup group1 = new ThreadGroup(base, "Group1");

ThreadGroup group2 = new ThreadGroup(base, "Group2");

The ThreadGroup class provides several convenient methods that work on all threads at once, here to name a few:

- activeCount(): returns an estimate of the number of active threads in the thread group and its subgroups.

- activeGroupCount(): returns an estimate of the number of active groups in the thread group and its subgroups.

- destroy(): destroys the thread group and all of its subgroups.

- enumerate(Thread[] list): copies into the specified array every active thread in this thread group and its subgroups.

- getMaxPriority(): returns the maximum priority of the thread group.

- interrupt(): interrupts all threads in the thread group.

- isDaemon(): tests if the thread group is a daemon thread group.

- setMaxPriority(int priority): sets the maximum priority of the group.

Let’s see an example in action. Consider the following class:

class Task extends Thread {

      public Task(ThreadGroup threadGroup, String name) {

            super(threadGroup, name);

      }

      public void run() {

            boolean running = true;

            while (running) {

                  try {

                        System.out.println(getName() + " is running");

                        Thread.sleep(1000);

                  } catch (InterruptedException ex) {

                        running = false;

                        System.out.println(getName() + " is interrupted and then terminates");

                  }

            }

      }

}

As you can see in the run() method, this thread will runs forever until it is interrupted by another thread. And here is the test program:

public class ThreadGroupExample {

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

            ThreadGroup group = new ThreadGroup("GroupA");

            new Task(group, "A").start();

            new Task(group, "B").start();

            new Task(group, "C").start();

            new Task(group, "D").start();

            Thread.sleep(10000);

            group.interrupt();

      }

}

As you can see, 4 threads are created and added to one thread group and they are all started to run concurrently. The main thread calls interrupt() on the group after the program has been running for 10 seconds:

Thread.sleep(10000);

group.interrupt();

This results in all threads in the group are interrupted, and by intention of the code, the threads terminate, hence the following output repeats 10 times:

A is running

D is running

B is running

C is running

and the last output looks like this:

C is interrupted and then terminates

B is interrupted and then terminates

A is interrupted and then terminates

D is interrupted and then terminates

Use ReadWriteLock

Basically, a ReadWriteLock is designed as a high-level locking mechanism that allows you to add thread-safety feature to a data structure while increasing throughput by allowing multiple threads to read the data concurrently and one thread to update the data exclusively.

ReadWriteLock is an interface defined in the java.util.concurrent.locks package, with ReentrantReadWriteLock is an implementation class. So you can create a ReadWriteLock like this:

ReadWriteLock rwLock = new ReentrantReadWriteLock();

The ReentrantReadWriteLock maintains two separate locks, one for reading and one for writing:

Lock readLock = rwLock.readLock();

Lock writeLock = rwLock.writeLock();

Then you can use the read lock to safeguard a code block that performs read operation like this:

readLock.lock();

try {

      // reading data

} finally {

      readLock.unlock();

}

And use the write lock to safeguard a code block that performs update operation like this:

writeLock.lock();

try {

      // update data

} finally {

      writeLock.unlock();

}

A ReadWriteLock implementation guarantees the following behaviors:

- Multiple threads can read the data at the same time, as long as there’s no thread is updating the data.

- Only one thread can update the data at a time, causing other threads (both readers and writers) block until the write lock is released.

- If a thread attempts to update the data while other threads are reading, the write thread also blocks until the read lock is released.

So ReadWriteLock can be used to add concurrency features to a data structure, but it doesn’t guarantee the performance because it depends on various factors: how the data structure is designed, the contention of reader and writer threads at real time, CPU architecture (single core  or multicores), etc.

Similar to ReentrantLock, the ReentrantReadWriteLock allows a thread to acquire the read lock or write lock multiple times recursively, thus the word “Reentrant”.

Let’s see an example that adds concurrency features to an ArrayList using ReentrantReadWriteLock. Consider the following data structure:

import java.util.\*;

import java.util.concurrent.locks.\*;

public class ReadWriteList<E> {

      private List<E> list = new ArrayList<>();

      private ReentrantReadWriteLock rwLock = new ReentrantReadWriteLock();

      public ReadWriteList(E... initialElements) {

            list.addAll(Arrays.asList(initialElements));

      }

      public void add(E element) {

            Lock writeLock = rwLock.writeLock();

            writeLock.lock();

            try {

                  list.add(element);

            } finally {

                  writeLock.unlock();

            }

      }

      public E get(int index) {

            Lock readLock = rwLock.readLock();

            readLock.lock();

            try {

                  return list.get(index);

            } finally {

                  readLock.unlock();

            }

      }

      public int size() {

            Lock readLock = rwLock.readLock();

            readLock.lock();

            try {

                  return list.size();

            } finally {

                  readLock.unlock();

            }

      }

}

As you can see, this class wraps an ArrayList as the underlying data structure. It uses the read lock to safeguard concurrent access to the read operations (get() and size() methods) and uses the write lock to safeguard concurrent access to the write operations (add() method).

Next, create a writer thread that randomly adds a number to the shared list. Here’s the code:

public class Writer extends Thread {

      private ReadWriteList<Integer> sharedList;

      public Writer(ReadWriteList<Integer> sharedList) {

            this.sharedList = sharedList;

      }

      public void run() {

            Random random = new Random();

            int number = random.nextInt(100);

            sharedList.add(number);

            try {

                  Thread.sleep(100);

                  System.out.println(getName() + " -> put: " + number);

            } catch (InterruptedException ie ) { ie.printStackTrace(); }

      }

}

Also create a reader thread that randomly gets an element from the shared list. Here’s the code:

public class Reader extends Thread {

      private ReadWriteList<Integer> sharedList;

      public Reader(ReadWriteList<Integer> sharedList) {

            this.sharedList = sharedList;

      }

      public void run() {

            Random random = new Random();

            int index = random.nextInt(sharedList.size());

            Integer number = sharedList.get(index);

            System.out.println(getName() + " -> get: " + number);

            try {

                  Thread.sleep(100);

            } catch (InterruptedException ie ) { ie.printStackTrace(); }

      }

}

Now, let’s create a test program like the following:

public class ReadWriteLockTest {

      static final int READER\_SIZE = 10;

      static final int WRITER\_SIZE = 2;

      public static void main(String[] args) {

            Integer[] initialElements = {33, 28, 86, 99};

            ReadWriteList<Integer> sharedList = new ReadWriteList<>(initialElements);

            for (int i = 0; i < WRITER\_SIZE; i++) {

                  new Writer(sharedList).start();

            }

            for (int i = 0; i < READER\_SIZE; i++) {

                  new Reader(sharedList).start();

            }

      }

}

As you can see, this program creates and runs 10 reader threads and 2 writer threads that work on a shared ReadWriteList data structure. Compile and run the program to observe the output.

So far I have shared with you the fundamentals and basic usage of ReadWriteLock with ReentrantReadWriteLock implementation. If you consult the Javadoc you will see that it is more sophisticated with fairness policy and other locking methods, so read more about it when you have time.

To conclude, remember the following key points:

- ReadWriteLock allows multiple concurrent readers abut only one exclusive writer.

- ReentrantReadWriteLock is an implementation of ReadWriteLock. In addition, it allows a reader/writer thread acquire a read lock/write lock multiple times recursively (reentrancy).

- Use the read lock to safeguard code that performs read operations, and use the write lock to protect access to code that performs update operation.

- In practice, ReadWriteLock can be used to increase throughput for shared data structure like cache or dictionary-like data which the update is infrequent and read is more frequent.

## Understanding Atomic Variables

Look at the java.util.concurrent.atomic package you will see the following classes:

      AtomicBoolean

      AtomicInteger

      AtomicLong

You can think of these are wrapper of primitive types boolean, integer and long, with the difference: they are designed to be safely used in multi-threaded context.

They are called atomic variables because they provide some operations that cannot be interfered by multiple threads. Here’s to name a few:

incrementAndGet(): Atomically increments by one the current value.

decrementAndGet(): Atomically decrements by one the current value.

These operations are guaranteed to execute atomically using machine-level instructions on modern processors.

Using atomic variables help avoiding the overhead of synchronization on a single primitive variable, so it is more efficient than using synchronization/locking mechanism.

To understand clearly, let’s walk through an example.

Consider the following Counter class:

public class Counter {

      private int value;

      public void increment() {

            value++;

      }

      public void decrement() {

            value--;

      }

      public int get() {

            return value;

      }

}

This class has two methods that update the value of an int variable, and a method to get the value.

Next, suppose we have a thread class that updates a shared instance of Counter like this:

public class UpdateThread extends Thread {

      private Counter counter;

      public UpdateThread(Counter counter) {

            this.counter = counter;

      }

      public void run() {

            try {

                  Thread.sleep(100);

            } catch (InterruptedException ex) { ex.printStackTrace(); }

            counter.increment();

      }

}

As you can see, this thread simply increases the value of the counter variable after sleeping for a short time (100 milliseconds).

And here’s the test program that runs 100 threads that concurrently update a shared instance of Counter:

public class ThreadsTest {

      static final int NUMBER\_THREADS = 100;

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

            Counter counter = new Counter();

            System.out.println("Initial Counter = " + counter.get());

            UpdateThread[] threads = new UpdateThread[NUMBER\_THREADS];

            for (int i = 0; i < NUMBER\_THREADS; i++) {

                  threads[i] = new UpdateThread(counter);

                  threads[i].start();

            }

            for (int i = 0; i < NUMBER\_THREADS; i++) {

                  threads[i].join();

            }

            System.out.println("Final Counter = " + counter.get());

      }

}

Let’s do a simple math. There are 100 threads, each increase the counter by one, so eventually the final value of the counter variable must be 100, right?

Now, try to compile and run this test program. You will see that sometimes it prints correct result:

Initial Counter = 0

Final Counter = 100

But more than one time, it prints incorrect result:

Initial Counter = 0

Final Counter = 96

The result is inconsistent. You can easily figure out why this happens, because the increment() method is executed by multiple threads concurrently without any synchronization or locking.

We can fix the problem by adding synchronization for the Counter class like this:

public class Counter {

      private int value;

      public synchronized void increment() {

            value++;

      }

      public synchronized void decrement() {

            value--;

      }

      public synchronized int get() {

            return value;

      }

}

or using explicit locking mechanism like this:

import java.util.concurrent.locks.\*;

public class Counter {

      private int value;

      private Lock lock = new ReentrantLock();

      public void increment() {

            lock.lock();

            value++;

            lock.unlock();

      }

      public void decrement() {

            lock.lock();

            value--;

            lock.unlock();

      }

      public synchronized int get() {

            return value;

      }

}

Now recompile and run the test program again for at least 10 times, you will realize that the problem has gone, proved by the consistent output:

Initial Counter = 0

Final Counter = 100

However, synchronization/locking comes at the cost of slow performance as it requires resources and thread scheduler to monitor the lock.

Therefore, atomic variable is a good alternative to synchronization on a single primitive type as mentioned earlier, atomic variable uses machine-level instructions to guarantee atomicity.

For example, you can use the AtomicInteger class to replace the int primitive type in the Counter class like this:

import java.util.concurrent.atomic.\*;

public class Counter {

      private AtomicInteger value = new AtomicInteger();

      public void increment() {

            value.incrementAndGet();

      }

      public void decrement() {

            value.decrementAndGet();

      }

      public int get() {

            return value.get();

      }

}

Here, the methods incrementAndGet() and decrementAndGet() guarantee to execute atomically, which means that they are safely executed by multiple threads.

Now recompile and run the test program again, you will observe the same result as using synchronization/locking with better performance though it’s hard to see the difference with this simple example. At least you got the idea, right?

In addition to increment/decrement methods, the AtomicInteger and AtomicLong classes provide other atomic methods such as:

- addAndGet(int delta): Atomically adds the given value to the current value.

- compareAndSet(int expect, int update): Atomically sets the value to the given updated value if the current value == the expected value.

- getAndAdd(int delta): Atomically adds the given value to the current value.

- set(int newValue): Sets to the given value.

Besides atomic variables for primitive types, the Java Concurrency API also provides atomic arrays and atomic reference type:

      - AtomicIntegerArray

      - AtomicLongArray

      - AtomicReference

      - AtomicReferenceArray

This classes allow programmers to perform atomic operations on arrays and reference types. Consult  Javadoc of the java.util.concurrent.atomic package for more information.

## Understand Parallel Programming with Fork/Join Framework

Today I’m going to help you understand and experiment with Fork/Join framework, which is used by several new features in Java 7 and Java 8. You will be able to write programs that *run tasks in parallel* utilized multicore processors which are very popular today (perhaps your computer’s CPU has at least 2 or 4 cores, doesn’t it?).

Notice that parallel execution is different than concurrent execution:

- In parallel execution, each thread is executed in a separate processing core. Therefore, tasks are really executed in true parallel fashion.

- In concurrent execution, the threads are executed on a same core. That means tasks are actually executed in interleave fashion, sharing processing time of a processing core.

Don’t worry if you think parallel programming is complex and difficult, as you will see the Fork/Join framework makes it easy for programmers.

Continue reading because parallel programming will be part of every programmer’s future.

**\* What is Fork/Join Framework?**

Fork/Join framework is a set of APIs that allow programmers to take advantage of parallel execution supported by multicore processors. It uses ‘divide-and-conquer’ strategy: divide a very large problem into smaller parts, which in turn, the small part can be divided further into smaller ones, recursively until a part can be solved directly. This is called ‘fork’.

Then all parts are executed in parallel on multiple processing cores. The results of each part are ‘joined’ together to produce the final result. Hence the name of the framework ‘Fork/Join’.

The following pseudo code illustrates how the divide and conquer strategies work with Fork/Join framework:

*if (problemSize < threshold)*

*solve problem directly*

*else {*

*break problem into subproblems*

*recursively solve each problem*

*combine the results*

*}*

Fork/Join framework is added to JDK since Java 7 and improved in Java 8. It is used by several new features in the Java programming language, including Streams API and sorting an array in parallel.

Fork/Join framework simplifies parallel programming because:

- It simplifies thread creation. Threads are created and managed automatically.

- It automatically makes use of multiple processors so programs can scale to make use of available processors.

With support for true parallel execution, Fork/Join framework can significantly reduce computation time and increase performance in solving very large problems such as image processing, video processing, big data processing, etc.

One interesting point about Fork/Join framework: it uses a *work stealing algorithm* to balance the load among threads: if a worker thread runs out of things to do, it can steal tasks from other threads that are still busy.

**\* Understand Fork/Join Framework’s API**

The Fork/Join framework API is implemented in the java.util.concurrent package. At its core are the following 4 classes:

- **ForkJoinTask**<V>: an abstract class that defines a task that runs within a ForkJoinPool.

- **ForkJoinPool**: a thread pool that manages the execution of ForkJoinTasks.

- **RecursiveAction**: a ForkJoinTask’s subclass for tasks that don’t return values.

- **RecursiveTask**<V>: a ForkJoinTask’s subclass for tasks that return values.

Basically, you implement code for solving problems in a subclass of either RecursiveAction or RecursiveTask. And then submit the task to be executed by a ForkJoinPool, which handles everything from threads management to utilization of multicore processor.

Let’s dive deeper into each of these classes before going through some code examples.

**ForkJoinTask<V>**

This is the abstract base class for tasks that run within a ForkJoinPool. The type parameter V specifies the result type of the task. A ForkJoinTask is a thread-like entity that represents lightweight abstraction of a task, rather than an actual thread of execution. This mechanism allows a large number o tasks to be managed by a small number of actual threads in a ForkJoinPool. Its key methods are:

* final ForkJoinTask<V> **fork**()
* final V **join**()
* final V **invoke**()

The **fork**() method submits the task to execute asynchronously. This method return this (ForkJoinTask) and the calling thread continues to run.

The **join**() method waits until the task is done and returns the result.

The **invoke**() method combines fork() and join() in a single call. It starts the task, waits for it to end and return the result.

In addition, the ForkJoinTask class provides a couple of static methods for invoking more than one task at a time:

* static void **invokeAll**(ForkJoinTask<?> task1, ForkJoinTask<?> task2): execute two tasks.
* static void **invokeAll**(ForkJoinTask<?>… taskList): execute a list of tasks.

**RecursiveAction:**

This is a recursive ForkJoinTask that doesn’t return a result. “Recursive” means that the task can be split into subtasks of itself by divide-and-conquer strategy (you’ll see how to divide in the code examples the next email).

You must override its abstract method compute() in which computational code is put.

protected abstract void **compute**();

**RecursiveTask<V>:**

Similar to RecursiveAction, but a RecursiveTask returns a result whose type is specified by the type parameter V. You also must to put computational code by overriding the compute() method:

protected abstract V **compute**();

**ForkJoinPool:**

This class is the heart of Fork/Join framework. It’s responsible for the management of threads and execution of ForkJoinTasks. You must first have an instance of ForkJoinPool in order to execute ForkJoinTasks.

There are two ways for acquiring a ForkJoinPool instance. The first way creates a ForkJoinPool object using one of its constructors:

* **ForkJoinPool**(): creates a default pool that supports a level of parallelism equal to the number of processors available in the system.
* **ForkJoinPool**(int parallelism): creates a pool with a custom level of parallelism which must be greater than 0 and not more than the actual number of processors available.

The level of parallelism determines the number of threads that can execute concurrently. In other words, it determines the number of tasks that can be executed simultaneously - but cannot exceed the number of processors.

However, that doesn’t limit the number of tasks that can be managed by the pool. A ForkJoinPool can manage many more tasks than its level of parallelism.

The second way to acquire a ForkJoinPool instance is obtaining the common pool instance using the following ForkJoinPool’s static method:

public static ForkJoinPool **commonPool**()

The common pool is statically constructed and automatically available for use.

**\* Execute ForkJoinTasks in a ForkJoinPool**

After you have created an instance of ForkJoinPool, you can start executing a task using one of the following methods:

* <T> T **invoke**(ForkJoinTask<T> task): executes the specified task and returns its result upon completion. This call is synchronous, meaning that the calling thread waits until this method returns. For a resultless task (RecursiveAction), the type parameter T is Void.
* void **execute**(ForkJoinTask<?> task): executes the specified task asynchronously - the calling code doesn’t wait for the task’s completion - it continues to run.

Alternatively, you can execute a ForkJoinTask by calling its own methods**fork**() or **invoke**(). In this case, the common pool will be used automatically, if the task is not already running within a ForkJoinPool.

A noteworthy point: ForkJoinPool uses daemon threads that are terminated when all user threads are terminated. That means you don’t have to explicitly shutdown a ForkJoinPool (though it is possible). In the case of common pool, calling shutdown() has no effect because the pool is always available for use.

## Fork/Join Framework Code Examples

**\* Example #1 - Using RecursiveAction**

In this first example, you will learn how to use the Fork/Join framework to execute a task that doesn’t return a result, by extending the **RecursiveAction**class.

Suppose that we need to do a transformation on a very large array of numbers. For the sake of simplicity, the transformation is simply multiply every element in the array by a specified number. The following code is for the transformation task:

import java.util.concurrent.\*;

public class ArrayTransform extends RecursiveAction {

      int[] array;

      int number;

      int threshold = 100\_000;

      int start;

      int end;

      public ArrayTransform(int[] array, int number, int start, int end) {

            this.array = array;

            this.number = number;

            this.start = start;

            this.end = end;

      }

      protected void compute() {

            if (end - start < threshold) {

                  computeDirectly();

            } else {

                  int middle = (end + start) / 2;

                  ArrayTransform subTask1 = new ArrayTransform(array, number, start, middle);

                  ArrayTransform subTask2 = new ArrayTransform(array, number, middle, end);

                  invokeAll(subTask1, subTask2);

            }

      }

      protected void computeDirectly() {

            for (int i = start; i < end; i++) {

                  array[i] = array[i] \* number;

            }

      }

}

As you can see, this is a subclass of RecursiveAction and it implements the computation in the compute() method.

The array and number are passed from its constructor. The parameters start and end specify the range of elements in the array to be processed. This helps splitting the array into sub arrays if its size is greater than a threshold, otherwise perform the computation on the whole array directly.

Look at the code snippet in the else block in the compute() method:

protected void compute() {

      if (end - start < threshold) {

            computeDirectly();

      } else {

            int middle = (end + start) / 2;

            ArrayTransform subTask1 = new ArrayTransform(array, number, start, middle);

            ArrayTransform subTask2 = new ArrayTransform(array, number, middle, end);

            invokeAll(subTask1, subTask2);

      }

}

Here we divide the array into 2 parts and create two subtasks that process each. In turn, the subtask may be also divided further into smaller subtasks recursively until the size is less than the threshold, which invokes the computeDirectly() method.

And then you can execute the main task on a ForkJoinPool like this:

ArrayTransform mainTask = new ArrayTransform(array, number, 0, SIZE);

ForkJoinPool pool = new ForkJoinPool();

pool.invoke(mainTask);

or execute the task on the common pool:

ArrayTransform mainTask = new ArrayTransform(array, number, 0, SIZE);

mainTask.invoke();

Here’s the full source code of the test program:

import java.util.\*;

import java.util.concurrent.\*;

public class ForkJoinRecursiveActionTest {

      static final int SIZE = 10\_000\_000;

      static int[] array = randomArray();

      public static void main(String[] args) {

            int number = 9;

            System.out.println("First 10 elements of the array before: ");

            print();

            ArrayTransform mainTask = new ArrayTransform(array, number, 0, SIZE);

            ForkJoinPool pool = new ForkJoinPool();

            pool.invoke(mainTask);

            System.out.println("First 10 elements of the array after: ");

            print();

      }

      static int[] randomArray() {

            int[] array = new int[SIZE];

            Random random = new Random();

            for (int i = 0; i < SIZE; i++) {

                  array[i] = random.nextInt(100);

            }

            return array;

      }

      static void print() {

            for (int i = 0; i < 10; i++) {

                  System.out.print(array[i] + ", ");

            }

            System.out.println();

      }

}

As you can see, we test with an array of 10 million elements that are randomly generated. As the array is too large, we print only the first 10 elements before and after the computation to see the effect:

First 10 elements of the array before:

42, 98, 43, 14, 9, 92, 33, 18, 18, 76,

First 10 elements of the array after:

378, 882, 387, 126, 81, 828, 297, 162, 162, 684,

**\* Example #2 - Using RecursiveTask**

In this second example, you will learn how to implement a task that returns a result. The following task counts the occurrences of even numbers in a large array:

import java.util.concurrent.\*;

public class ArrayCounter extends RecursiveTask<Integer> {

      int[] array;

      int threshold = 100\_000;

      int start;

      int end;

      public ArrayCounter(int[] array, int start, int end) {

            this.array = array;

            this.start = start;

            this.end = end;

      }

      protected Integer compute() {

            if (end - start < threshold) {

                  return computeDirectly();

            } else {

                  int middle = (end + start) / 2;

                  ArrayCounter subTask1 = new ArrayCounter(array, start, middle);

                  ArrayCounter subTask2 = new ArrayCounter(array, middle, end);

                  invokeAll(subTask1, subTask2);

                  return subTask1.join() + subTask2.join();

            }

      }

      protected Integer computeDirectly() {

            Integer count = 0;

            for (int i = start; i < end; i++) {

                  if (array[i] % 2 == 0) {

                        count++;

                  }

            }

            return count;

      }

}

As you can see, this class extends the RecursiveTask and overrides the compute() method that returns a result (an Integer in this case).

And note that we use the join() method to combine the results of subtasks:

return subTask1.join() + subTask2.join();

The test program is similar to the RecursiveAction example:

import java.util.\*;

import java.util.concurrent.\*;

public class ForkJoinRecursiveTaskTest {

      static final int SIZE = 10\_000\_000;

      static int[] array = randomArray();

      public static void main(String[] args) {

            ArrayCounter mainTask = new ArrayCounter(array, 0, SIZE);

            ForkJoinPool pool = new ForkJoinPool();

            Integer evenNumberCount = pool.invoke(mainTask);

            System.out.println("Number of even numbers: " + evenNumberCount);

      }

      static int[] randomArray() {

            int[] array = new int[SIZE];

            Random random = new Random();

            for (int i = 0; i < SIZE; i++) {

                  array[i] = random.nextInt(100);

            }

            return array;

      }

}

Run this program and you will see the output something like this:

Number of even numbers: 5000045

**\* Example #3 - Experiment with Parallelism**

In this last example, you will learn how the level of parallelism affects the computation time.

The ArrayCounter class is rewritten to have the threshold passed from constructor like this:

import java.util.concurrent.\*;

public class ArrayCounter extends RecursiveTask<Integer> {

      int[] array;

      int threshold;

      int start;

      int end;

      public ArrayCounter(int[] array, int start, int end, int threshold) {

            this.array = array;

            this.start = start;

            this.end = end;

            this.threshold = threshold;

      }

      protected Integer compute() {

            if (end - start < threshold) {

                  return computeDirectly();

            } else {

                  int middle = (end + start) / 2;

                  ArrayCounter subTask1 = new ArrayCounter(array, start, middle, threshold);

                  ArrayCounter subTask2 = new ArrayCounter(array, middle, end, threshold);

                  invokeAll(subTask1, subTask2);

                  return subTask1.join() + subTask2.join();

            }

      }

      protected Integer computeDirectly() {

            Integer count = 0;

            for (int i = start; i < end; i++) {

                  if (array[i] % 2 == 0) {

                        count++;

                  }

            }

            return count;

      }

}

And in the test program, the level of parallelism and threshold are passed as arguments to the program:

import java.util.\*;

import java.util.concurrent.\*;

public class ParallelismTest {

      static final int SIZE = 10\_000\_000;

      static int[] array = randomArray();

      public static void main(String[] args) {

            int threshold = Integer.parseInt(args[0]);

            int parallelism = Integer.parseInt(args[1]);

            long startTime = System.currentTimeMillis();

            ArrayCounter mainTask = new ArrayCounter(array, 0, SIZE, threshold);

            ForkJoinPool pool = new ForkJoinPool(parallelism);

            Integer evenNumberCount = pool.invoke(mainTask);

            long endTime = System.currentTimeMillis();

            System.out.println("Number of even numbers: " + evenNumberCount);

            long time = (endTime - startTime);

            System.out.println("Execution time: " + time + " ms");

      }

      static int[] randomArray() {

            int[] array = new int[SIZE];

            Random random = new Random();

            for (int i = 0; i < SIZE; i++) {

                  array[i] = random.nextInt(100);

            }

            return array;

      }

}

This program allows you to easily test the performance with different values of parallelism and threshold. Note that it prints the execution time at the end. Try to run this program several times with different arguments and observe the execution time. Here are the suggested commands:

java ParallelismTest 1 100000

java ParallelismTest 2 100000

java ParallelismTest 3 100000

java ParallelismTest 4 100000

java ParallelismTest 2 500000

java ParallelismTest 4 500000

…

**\* Conclusion**

So far I have walked you through a lesson about Fork/Join framework. Here are the key points to remember:

- Fork/Join framework is designed to simplify parallel programming for Java programmers.

- ForkJoinPool is the heart of Fork/Join framework. It allows many ForkJoinTasks to be executed by a small number of actual threads, with each thread running on a separate processing core.

- You can obtain an instance of ForkJoinPool by either using its constructor or static method commonPool() that returns the common pool.

- ForkJoinTask is an abstract class that represents a task that is lighter weight than a normal thread. You implement the computation logic by overriding its compute() method.

- RecursiveAction is a ForkJoinTask that doesn’t return a result.

- RecursiveTask is a ForkJoinTask that returns a result.

- ForkJoinPool is different than other pools as it uses work stealing algorithm which allows a thread that runs out of things to do, to steal tasks from other threads that are still busy.

- Threads in ForkJoinPool are daemon. You don’t have to explicitly shutdown the pool.

- You can execute a ForkJoinTask either by invoking its own methods invoke() or fork(), or by submitting the task to a ForkJoinPool and then call invoke() or execute() on the pool.

- Calling invoke() or fork() on a ForkJoinTask will cause the task to run in the common pool, if it is not already running in a ForkJoinPool.

- Use the join() method on ForkJoinTasks to combine the results.

- The invoke() method waits for the task’s completion, but the execute() method does not.

Concurrency API in Java

In the previous lessons you got familiar with concurrent programming by using the low-level API such as working with threads (create, start, pause, stop), using locks and synchronized keyword. That’s good for the basics and simple usages.

For more advanced tasks, using the low-level API is time-consuming and error-prone. That’s why the high-level concurrency API is designed to help programmers easily implement more complex multi-threading tasks. Programmers can focus on the business logic of the tasks rather than getting busy in the low-level details.

The high-level concurrency API is implemented in the following 3 packages:

* **java.util.concurrent**: provides utility classes commonly useful in concurrent programming such as executors, threads pool management, scheduled tasks execution, the Fork/Join framework, concurrent collections, etc.
* **java.util.concurrent.locks**: provides Lock and Condition implementations that are more advanced than the built-in locking and synchronization mechanism.
* **java.util.concurrent.atomic**: provides data type classes that are safely updated without using locks. For example, an AtomicInteger can be atomically incremented or decremented so you can use it as a shared variable without synchronization.

In the previous lessons, you learned how to use Lock and Condition in the java.util.concurrent.locks package. I will talk about atomic classes in the next lesson, and today I focus on helping you get familiar with the concurrent utilities in the java.util.concurrent package.

With the support of high-level concurrency API, you can do the following things (but not limited to):

* Executing tasks by multiple threads that are managed in a thread pool. You don’t have to manage the thread pool yourself, just choose a kind of pool you want and submit the tasks. This can be done via various implementations of ***Executor***.
* Queuing tasks to be executed sequentially, one after another.
* Running a task that computes a value (Callable) and waiting for the result (Future). This can be done by using an ExecutorService.
* Scheduling a task to be executed after a given delay, or to be executed periodically at a fixed rate or fixed delay. This can be done by using the ScheduleExecutorService with ScheduleFuture.
* Talking the advantages of multiple processors to perform heavy work faster by breaking the work into smaller pieces recursively. This can be done with the support of the Fork/Join framework.

**\* Understanding Thread Pool:**

In terms of performance, creating a new thread is an expensive operation because it requires the operating system allocates resources need for the thread. Therefore, in practice thread pool is used for large-scale applications that launch a lot of short-lived threads in order to utilize resources efficiently and increase performance.

Instead of creating new threads when new tasks arrive, a thread pool keeps a number of idle threads that are ready for executing tasks as needed. After a thread completes execution of a task, it does not die. Instead it remains idle in the pool waiting to be chosen for executing new tasks.

You can limit a definite number of concurrent threads in the pool, which is useful to prevent overload. If all threads are busily executing tasks, new tasks are placed in a queue, waiting for a thread becomes available.

That’s basically how thread pool works. In practice, thread pool is used widely in web servers where a thread pool is used to serve client’s requests. Thread pool is also used in database applications where a pool of threads maintaining open connections with the database.

Implementing a thread pool is a complex task, but you don’t have to do it yourself. As the Java Concurrency API allows you to easily create and use thread pools without worrying about the details.

**\* Understanding Executors:**

An Executor is an object that is responsible for threads management and execution of Runnable tasks submitted from the client code. It decouples the details of thread creation, scheduling, etc from the task submission so you can focus on developing the task’s business logic without caring about the thread management details.

That means, in the simplest case, rather than creating a thread to execute a task like this:

Thread t = new Thread(new RunnableTask());

t.start();

You submit tasks to an executor like this:

      Executor executor = anExecutorImplementation;

      executor.execute(new RunnableTask1());

      executor.execute(new RunnableTask2());

The Java Concurrency API defines the following 3 base interfaces for executors:

* **Executor**: is the super type of all executors. It defines only one method execute(Runnable).
* **ExecutorService**: is an Executor that allows tracking progress of value-returning tasks (Callable) via Future object, and manages the termination of threads. Its key methods include submit() and shutdown().
* **ScheduledExecutorService**: is an ExecutorService that can schedule tasks to execute after a given delay, or to execute periodically. Its key methods are schedule(), scheduleAtFixedRate() and scheduleWithFixedDelay().

You can create an executor by using one of several factory methods provided by the Executors utility class. Here’s to name a few:

* **newCachedThreadPool()**: creates an expandable thread pool executor. New threads are created as needed, and previously constructed threads are reused when they are available. Idle threads are kept in the pool for one minute. This executor is suitable for applications that launch many short-lived concurrent tasks.
* **newFixedThreadPool(int n)**: creates an executor with a fixed number of threads in the pool. This executor ensures that there are no more than n concurrent threads at any time. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread becomes available. If any thread terminates due to failure during execution, it will be replaced by a new one. The threads in the pool will exist until it is explicitly shutdown. Use this executor if you and to limit the maximum number of concurrent threads.
* **newSingleThreadExecutor()**: creates an executor that executes a single task at a time. Submitted tasks are guaranteed to execute sequentially, and no more than one task will be active at any time. Consider using this executor if you want to queue tasks to be executed in order, one after another.
* **newScheduledThreadPool(int corePoolSize)**: creates an executor that can schedule tasks to execute after a given delay, or to execute periodically. Consider using this executor if you want to schedule tasks to execute concurrently.
* **newSingleThreadScheduleExecutor()**: creates a single-threaded executor that can schedule tasks to execute after a given delay, or to execute periodically. Consider using this executor if you want to schedule tasks to execute sequentially.

In case the factory methods do not meet your need, you can construct an executor directly as an instance of either ThreadPoolExecutor or ScheduledThreadPoolExecutor, which gives you additional options such as pool size, on-demand construction, keep-alive times, etc.

**\* A Simple Executor Example:**

The following code snippet shows you a simple example of executing a task by a single-threaded executor:

import java.util.concurrent.\*;

public class SimpleExecutorExample {

      public static void main(String[] args) {

            ExecutorService pool = Executors.newSingleThreadExecutor();

            Runnable task = new Runnable() {

                  public void run() {

                        System.out.println(Thread.currentThread().getName());

                  }

            };

            pool.execute(task);

            pool.shutdown();

      }

}

As you can see, a Runnable task is created using anonymous-class syntax. The task simply prints the thread name and terminates. Compile and run this program and you will see the output something like this:

pool-1-thread-1

Note that you should call shutdown() to destroy the executor after the thread completes execution. Otherwise, the program is still running afterward. You can observe this behavior by commenting the call to shutdown.

Let’s see a more complex example in which I show you how to execute multiple tasks using different kinds of executors.

**\* Using a cached thread pool executor:**

Given the following class:

public class CountDownClock extends Thread {

      private String clockName;

      public CountDownClock(String clockName) {

            this.clockName = clockName;

      }

      public void run() {

            String threadName = Thread.currentThread().getName();

            for (int i = 5; i >= 0; i--) {

                  System.out.printf("%s -> %s: %d\n", threadName, clockName, i);

                  try {

                        Thread.sleep(1000);

                  } catch (InterruptedException ex) {

                        ex.printStackTrace();

                  }

            }

      }

}

This class represents a countdown clock that counts a number from 5 down to 0, and pause 1 second after every count. Upon running, it prints the current thread name, follows by the clock name and the count number.

Let’s create an executor with a cached thread pool to execute 4 clocks concurrently. Here’s the code:

import java.util.concurrent.\*;

public class MultipleTasksExecutorExample {

      public static void main(String[] args) {

            ExecutorService pool = Executors.newCachedThreadPool();

            pool.execute(new CountDownClock("A"));

            pool.execute(new CountDownClock("B"));

            pool.execute(new CountDownClock("C"));

            pool.execute(new CountDownClock("D"));

            pool.shutdown();

      }

}

Compile and run this program, you will see that there are 4 threads executing the 4 clocks concurrently:  
  
A screenshot of a computer screen

Description automatically generated

Modify this program to add more tasks e.g. add more 3 clocks. Recompile and run the program again, you will see that the number of threads is as equal as the number of submitted tasks. That’s the key behavior of a cached thread pool: new threads are created as needed.

**\* Using a fixed thread pool executor:**

Next, update the statement that creates the executor to use a fixed thread pool:

ExecutorService pool = Executors.newFixedThreadPool(2);

Here, we create an executor with a pool of maximum 2 concurrent threads. Keep only 4 task (4 clocks) submitted to the executor. Recompile and run the program you will see that there are only 2 threads executing the clocks:  
  
A screen shot of a computer

Description automatically generated

The clocks A and B run first, while the clocks C and D are waiting in the queue. After A and B completes execution, the 2 threads continue executing the clocks C and D. That’s the key behavior of a fixed thread pool: limiting the number of concurrent threads and queuing additional tasks.

**\* Using a single-threaded pool executor:**

Let’s update the program above to use a single-threaded executor like this:

      ExecutorService pool = Executors.newSingleThreadExecutor();

Recompile and run the program, you will see that there’s only one thread executing the 4 clocks sequentially:  
  
A screenshot of a computer

Description automatically generated

That’s the key behavior of a single-threaded executor: queue tasks to execute in order, one after another.

## Executing Value-Returning Tasks with Callable and Future

So far we have executed tasks that do not return any value (void). How about tasks those compute and return a value upon completion? Those tasks may take long time to finish, and what if we want to wait for the results?

The ExecutorService interface defines a method that allows us to execute such kind of task:

**< T > Future< T >   submit(Callable< T > task)**

Here, the type parameter **T** is the return type of the task. You submit a task that implements the **Callable** interface which defines only one method as follows:

public interface Callable< T > {

      public T call();

}

The purpose of the Callable interface is similar to Runnable, but its method returns a value of type T.

Once the task is submitted, the executor immediately returns an object of type **Future** representing the pending results of the task, for example:

      Callable< Integer > task = new task that returns an Integer value;

Future< Integer > result = executor.submit(task);

Then you can invoke the Future’s get() method to obtain the result upon successful completion. There are two overloads of this method defined as follows:

public interface Future< T > {

      T get();

      T get(long timeout, TimeUnit unit);

}

The first overload version waits if necessary for the computation to complete and then retrieves its result:

      Integer value = result.get();

This method blocks the current thread to wait until the computation completes and returns the value. In case you want to wait only for a specified amount of time, use the second overload version:

                Integer value = result.get(2, TimeUnit.SECONDS);

This call wais if necessary for at most 2 seconds for the computation to complete, and then retrieves the result if available. If the task takes longer time to complete, the call returns null.

Now, let’s see a complete example.

Suppose that we have two tasks: the first calculates the factorial value of N numbers, and the second computes the sum of N numbers.

Implementing the Callable interface, the first task is written as follows:

import java.util.concurrent.\*;

public class FactorialCalculator implements Callable< Integer > {

      private int n;

      public FactorialCalculator(int n) {

            this.n = n;

      }

      public Integer call() {

            int result = 1;

            for (int i = 1; i <= n; i++) {

                  result = result \* i;

            }

            try {

                  Thread.sleep(5000);

            } catch (InterruptedException ex) {

                  ex.printStackTrace();

            }

            return result;

      }

}

Here we use the sleep() method to fake the computation time. And code for the second task:

import java.util.concurrent.\*;

public class SumCalculator implements Callable< Integer > {

      private int n;

      public SumCalculator(int n) {

            this.n = n;

      }

      public Integer call() {

            int sum = 0;

            for (int i = 1; i <= n; i++) {

                  sum += i;

            }

            try {

                  Thread.sleep(2000);

            } catch (InterruptedException ex) {

                  ex.printStackTrace();

            }

            return sum;

      }

}

The following program submits two tasks above to a fixed thread pool executor:

import java.util.concurrent.\*;

public class CallableAndFutureExample {

      public static void main(String[] args) {

            ExecutorService pool = Executors.newFixedThreadPool(2);

            Future< Integer > sumResult = pool.submit(new SumCalculator(100000));

            Future< Integer > factorialResult = pool.submit(new FactorialCalculator(8));

            try {

                  Integer sumValue = sumResult.get();

                  System.out.println("Sum Value = " + sumValue);

                  Integer factorialValue = factorialResult.get();

                  System.out.println("Factorial Value = " + factorialValue);

            } catch (InterruptedException | ExecutionException ex) {

                  ex.printStackTrace();

            }

            pool.shutdown();

      }

}

Run this program and observe the result. The first task, SumCalculator takes 2 seconds to complete and you see the result displayed after 2 seconds:

Sum Value = 705082704

The second task, FactorialCalculator takes 5 seconds to complete, so you see the result 3 seconds after the first result:

Factorial Value = 40320

In addition, the Future interface provides methods for checking the completion status:  
  
  
  
It’s your exercise to experiment with these additional methods.

* boolean isDone(): returns true if this task completed.
* boolean isCancelled(): returns true if this task was cancelled before it completed normally.

and for stopping the task:

* boolean cancel(boolean mayInterruptIfRunning): attempts to cancel execution of this task. Returns false if the task could not be cancelled, typically because it has already completed normally; true otherwise.

Scheduling Tasks to Execute After a Delay or Periodically

Implementations of the ScheduledExecutorService interface provide convenient methods for scheduling tasks:

* **schedule**(Callable callable, long delay, TimeUnit unit): executes a Callable task after the specified delay. TimeUnit can be in MILLISECONDS, SECONDS, MINUTES, HOURS, etc.
* **schedule**(Runnable command, long delay, TimeUnit unit): Executes a Runnable task after a given delay.
* **scheduleAtFixedRate**(Runnable command, long initialDelay, long delay, TimeUnit unit): Executes a periodic task after an initial delay, then repeat after every given period. If any execution of this task takes longer than its period, then subsequent executions may start late, but will not concurrently execute.
* **scheduleWithFixedDelay**(Runnable command, long initialDelay, long delay, TimeUnit unit): Executes a periodic task after an initial delay, then repeat after every given delay between the termination of one execution and the commencement of the next.

All these methods return a ScheduleFuture object which is a Future with an additional method for checking the remaining delay time:

long getDelay(TimeUnit unit)

And as shown previously, a ScheduledExecutorService object can be created via factory methods of the Executors utility class:

* newScheduledThreadPool(int poolSize): creates a thread pool that can schedule tasks to execute concurrently.
* newSingleThreadScheduledExecutor(): creates a single-threaded executor that can schedule tasks to execute sequentially.

Now, let’s see some examples.

This is the simplest example that executes a task after 5 seconds:

import java.util.concurrent.\*;

public class SimpleScheduledExecutorExample {

      public static void main(String[] args) {

            ScheduledExecutorService scheduler = Executors.newSingleThreadScheduledExecutor();

            Runnable task = new Runnable() {

                  public void run() {

                        System.out.println("Hi!");

                  }

            };

            scheduler.schedule(task, 5, TimeUnit.SECONDS);

            scheduler.shutdown();

      }

}

As you can see, this program simply prints the message “Hi!” after a delay of 5 seconds, and terminates.

Next, the following program plays a sound ‘beep’ for every 2 seconds:

import java.util.concurrent.\*;

public class BeepClock implements Runnable {

      public void run() {

            System.out.print("\007");

      }

      public static void main(String[] args) {

            ScheduledExecutorService scheduler = Executors.newSingleThreadScheduledExecutor();

            scheduler.scheduleAtFixedRate(new BeepClock(), 4, 2, TimeUnit.SECONDS);

      }

}

Notice that, with the execution of periodic tasks, do not call shutdown() on the executor because it causes the program to terminate immediately.

The following is a more complex example that uses a pool of 3 threads to schedule 3 count down clocks to execute concurrently:

import java.util.concurrent.\*;

public class ConcurrentScheduleTasksExample {

      public static void main(String[] args) {

            ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(3);

            CountDownClock clock1 = new CountDownClock("A");

            CountDownClock clock2 = new CountDownClock("B");

            CountDownClock clock3 = new CountDownClock("C");

            scheduler.scheduleWithFixedDelay(clock1, 3, 10, TimeUnit.SECONDS);

            scheduler.scheduleWithFixedDelay(clock2, 3, 15, TimeUnit.SECONDS);

            scheduler.scheduleWithFixedDelay(clock3, 3, 20, TimeUnit.SECONDS);

      }

}

Here, you can see 3 clocks A, B and C are scheduled to start at the same time, after an initial delay of 3 seconds, but their periodic delay times are different. The following screenshot shows output of this program:  
  
A screen shot of a computer

Description automatically generated

You can use the returned ScheduleFuture object to cancel the tasks. This updated version of the program above stops the 3 clocks after 2 minutes, by using another schedule task:

import java.util.concurrent.\*;

public class ConcurrentScheduleTasksExample {

      public static void main(String[] args) {

            ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(3);

            CountDownClock clock1 = new CountDownClock("A");

            CountDownClock clock2 = new CountDownClock("B");

            CountDownClock clock3 = new CountDownClock("C");

            Future< ? > f1 = scheduler.scheduleWithFixedDelay(clock1, 3, 10, TimeUnit.SECONDS);

            Future< ? > f2 = scheduler.scheduleWithFixedDelay(clock2, 3, 15, TimeUnit.SECONDS);

            Future< ? > f3 = scheduler.scheduleWithFixedDelay(clock3, 3, 20, TimeUnit.SECONDS);

            Runnable cancelTask = new Runnable() {

                  public void run() {

                        f1.cancel(true);

                        f2.cancel(true);

                        f3.cancel(true);

                  }

            };

            scheduler.schedule(cancelTask, 120, TimeUnit.SECONDS);

      }

}

Recompile and run the program again and observe the result.

**\* Summary:**

Here I summarize the key points you have learned today:

* Know what you can do with the high-level concurrency API.
* Understand the need of thread pool and how it works.
* Understand 3 types of executors: Executor, ExecutorService and ScheduledExecutorService.
* Know how to create different kinds of thread pools via several factory methods of the Executors utility class.
* Understand how to execute value-returning tasks with Callable and Future.
* Know how to schedule tasks to execute after a given delay, or execute periodically after a fixed rate or delay.

## Concurrent Collection: Understanding CopyOnWriteArrayList

**\* Why CopyOnWriteArrayList ?**

Basically, a CopyOnWriteArrayList is similar to an ArrayList, with some additional and more advanced thread-safe features.

You know, ArrayList is not thread-safe so it’s not safe to use in multi-threaded applications. We can achieve thread-safe feature for an ArrayList by using a synchronized wrapper like this:

List< String > unsafeList = new ArrayList< >();

List< String > safeList = Collections.synchronizedList(unsafeList);

safeList.add("Boom");   // safe to use with multiple threads

However, this synchronized list has a limitation: all of its read and write methods (add, set, remove, iterator, etc) are synchronized on the list object itself. That means if a thread is executing add() method, it blocks other threads which want to get the iterator to access elements in the list, for example. Also, only one thread can iterate the list’s elements at a time, which can be inefficient. That’s quite rigid.

What if we want a more flexible collection which allows:

* Multiple threads executing read operations concurrently.
* One thread executing read operation and another executing write operation concurrently.
* Only one thread can execute write operation while other threads can execute read operations simultaneously.

The CopyOnWriteArrayList class is designed to enable such sequential write and concurrent reads features. For example, we can write a multi-threaded program that allows one thread to add elements to the list while other threads are traversing the list’s elements at the same time, and no worry about ConcurrentModificationException as per the case of a synchronized list.

That’s interesting. So how does the CopyOnWriteArrayList implement this concurrent feature?

**\* How does CopyOnWriteArrayList works?**

The CopyOnWriteArrayList class uses a mechanism called ***copy-on-write***which works like this: For every write operation (add, set, remove, etc) it makes a new copy of the elements in the list. That means the read operations (get, iterator, listIterator, etc) work on a different copy.

In addition, a thread must acquire a separate lock before executing a write operation, and all write operations use this same lock so there’s only one write operation can be executed by only one thread at a time. The read operations do not use any lock so multiple threads can execute multiple read operations simultaneously. And of course, read and write operations do not block each other.

The methods iterator() and listIterator() return an iterator object that holds different copy of the elements, hence the term ***snapshot iterator***. The snapshot iterator doesn’t allow modifying the list while traversing, and it will not throw ConcurrentModificationException if the list is being modified by other thread during the traversal, and the read and write operations work on different copies of elements.

**\* When to Use CopyOnWriteArrayList?**

Due to its special behaviors, CopyOnWriteArrayList is suitable for use in scenarios require sequential write and concurrent reads on a same collection. But you should take performance issue into consideration because the process of copying elements is costly for a list that has a large number of elements and many write operations.

Having said that, use CopyOnWriteArrayList only when the number of write operations is very small as compared to the read operations and the list contains a small number of elements.

In some cases, we can use CopyOnWriteArrayList as a thread-safe alternative to ArrayList, and to take advantages of its new methods addIfAbsent() and addAllAbsent(), which are explained below.

**\* Understanding CopyOnWriteArrayList API:**

CopyOnWriteArrayList is a member of the Java Collection framework and is an implementation the List interface so it has all typical behaviors of a list. CopyOnWriteArrayList is considered as a thread-safe alternative to ArrayList with some differences:

- You can pass an array when creating a new CopyOnWriteArrayList object. The list holds a copy of this array, for example:

String[] fruits = {"Apple", "Banana", "Lemon", "Grape", "Mango"};

List< String > list = new CopyOnWriteArrayList< >(fruits);

- Though a list allows duplicate elements, you can add an element to the list if and only if it is not already in the list, by using the method addIfAbsent(element). More importantly, this method is thread-safe which means it guarantees no other threads can add the same element at the same time. This method returns true if the element was added. For example:

CopyOnWriteArrayList< String > list = new CopyOnWriteArrayList< >();

list.add("Apple");

list.add("Banana");

if (list.addIfAbsent("Orange")) {

      System.out.println("Orange was added");

}

- Similarly, the method addAllAbsent(Collection) appends all elements in the specified collection that are not already contained in the list. And more importantly, this method is thread-safe. This method returns the number of elements were added. For example:

CopyOnWriteArrayList< String > list1 = new CopyOnWriteArrayList< >();

list1.add("Apple");

list1.add("Banana");

List< String > list2 = Arrays.asList("Lemon", "Banana");

int result = list1.addAllAbsent(list2);

System.out.println("Elements added: " + result);

This print Elements added: 1 because the element Banana is already contained in the list1.

 - The method iterator() returns a generic Iterator that holds a snapshot of the list. This iterator doesn’t support the remove() method.

- The method listIerator() returns a generic ListIterator that holds a snapshot of the list. This iterator doesn’t support the remove(), set() or add()  method.

 And as stated previously, the iterator will not throw ConcurrentModificationException if the list is being modified by another thread while the current thread is traversing the iterator, because a snapshot iterator holds a different copy of elements.

**\* CopyOnWriteArrayList Examples:**

Let’s see a couple of examples in action. The first one creates two threads:

- Thread Writer adds a number to CopyOnWriteArrayList for every 5 seconds.

- Thread Reader iterates the list repeatedly with a small delay (10 milliseconds) for every iteration.

That means the read operations outnumber the write ones, and here’s the full source code of the program:

import java.util.\*;

import java.util.concurrent.\*;

public class CopyOnWriteArrayListTest {

      public static void main(String[] args) {

            List< Integer > list = new CopyOnWriteArrayList< >();

            list.add(1);

            list.add(2);

            list.add(3);

            list.add(4);

            list.add(5);

            new WriteThread("Writer", list).start();

            new ReadThread("Reader", list).start();

      }

}

class WriteThread extends Thread {

      private List< Integer > list;

      public WriteThread(String name, List< Integer > list) {

            this.list = list;

            super.setName(name);

      }

      public void run() {

            int count = 6;

            while (true) {

                  try {

                        Thread.sleep(5000);

                  } catch (InterruptedException ex) {

                        ex.printStackTrace();

                  }

                  list.add(count++);

                  System.out.println(super.getName() + " done");

            }

      }

}

class ReadThread extends Thread {

      private List< Integer > list;

      public ReadThread(String name, List< Integer > list) {

            this.list = list;

            super.setName(name);

      }

      public void run() {

            while (true) {

                  String output = "\n" + super.getName() + ":";

                  Iterator< Integer > iterator = list.iterator();

                  while (iterator.hasNext()) {

                        Integer next = iterator.next();

                        output += " " + next;

                        // fake processing time

                        try {

                              Thread.sleep(10);

                        } catch (InterruptedException ex) {

                              ex.printStackTrace();

                        }

                  }

                  System.out.println(output);

            }

      }

}

Run this program and observe the result. You will see that the reader thread constantly prints out the elements in the list, whereas the writer thread slowly adds a new number to the list. This program runs forever until you press Ctrl + C to stop it.

Try to change the list implementation from CopyOnWriteArrayList to ArrayList like this:

List< Integer > list = new ArrayList< >();

Recompile and run the program again, you will see that the reader thread throws ConcurrentModificationException as soon as the writer thread adds a new element to the list. The reader thread die and only the writer thread alive.

The second example demonstrates how to use CopyOnWriteArrayList in event handling. Consider the following class:

public class GuiComponent {

      private List< ActionListener > listeners = new CopyOnWriteArrayList< >();

      public void addActionListener(ActionListener listener) {

            listeners.add(listener);

      }

      public void removeActionListener(ActionListener listener) {

            listeners.remove(listener);

      }

      public void fireActionEvent() {

            for (ActionListener listener : listeners) {

                  listener.actionPerformed(new ActionEvent(this, "message"));

            }

      }

}

Suppose this class represents a GUI component which can receives events such as mouse click. The client code can register (subscribe) to receive notification when the event occurs via the following method:

public void addActionListener(ActionListener listener)

The components use a CopyOnWriteArrayList object to maintain all registered listeners:

private List< ActionListener > listeners = new CopyOnWriteArrayList< >();

When an event occurs, the component notifies all of its listeners by iterating the list and invoke the action handler method on each listener, as shown in the fireActionEvent() method:

for (ActionListener listener : listeners) {

      listener.actionPerformed(new ActionEvent(this, "message"));

}

The ActionListener interface is defined as follows:

public interface ActionListener {

      public void actionPerformed(ActionEvent evt);

}

And the ActionEvent class is implemented as follows:

public class ActionEvent {

      Object source;

      Object data;

      public ActionEvent(Object source, Object data) {

            this.source = source;

            this.data = data;

      }

}

Look at the GuiComponent class, a CopyOnWriteArrayList is used because:

- It allows two threads can both read and write the list concurrently: one thread adds or removes a listener and the other thread notifies all listeners.

- The number of listeners is changed infrequently, whereas the number of times the listeners are notified more frequently (the read operations outnumber the write ones).

- It doesn’t throw ConcurrentModificationException if a thread is adding/removing a listener while another thread is iterating the list of listeners.

**Summary:**

CopyOnWriteArrayList can be used as a thread-safe alternative to ArrayList, with additional methods addIfAbsent() and addAllAbsent() that append elements if they are not contained in the list. A CopyOnWriteArrayList makes a new copy of its elements for every write operation and its iterator holds a different copy (snapshot) so it enables sequential writes and concurrent reads: only one thread can execute write operation and multiple threads can execute read operations at the same time. And its iterator doesn’t throw ConcurrentModification.

## Understanding CopyOnWriteArraySet

The second concurrent collection in the java.util.concurrent package I’d like to talk about is CopyOnWriteArraySet, which is similar to CopyOnWriteArrayList. Actually, a CopyOnWriteArraySet uses a CopyOnWriteArrayList internally for its operations. Thus it has the following behaviors:

* It’s thread-safe, and can be used as a thread-safe alternative to HashSet.
* It allows sequential write and concurrent reads by multiple threads. Only one thread can execute write operations (add and remove), and multiple threads can execute read operations (iterator) at a time.
* Its iterator doesn’t throw ConcurrentModificationException and doesn’t support remove method.

CopyOnWriteArraySet is a Set so it doesn’t allow duplicate elements. But unlike HashSet, its iterator returns elements in the order they were added.

Therefore, consider using a CopyOnWriteArraySet if you need a thread-safe collection that is similar to CopyOnWriteArrayList but no duplicate elements are allowed.

Also note that you should use CopyOnWriteArraySet only when the read operations outnumber the write operations and has a small number of elements, because the set creates a new copy of its elements for each write operation, which affects performance if the set has a large number of elements and the write operations are frequent.

Let’s see an example that demonstrates how  CopyOnWriteArraySet works. The following program e creates two threads:

- Thread Writer adds a number to CopyOnWriteArraySet for every 5 seconds.

- Thread Reader iterates the list repeatedly with a small delay (10 milliseconds) for every iteration.

Here’s the full source code of the program:

import java.util.\*;

import java.util.concurrent.\*;

public class CopyOnWriteArraySetTest {

      public static void main(String[] args) {

            Set< Integer > set = new CopyOnWriteArraySet< >();

            set.add(1);

            set.add(2);

            set.add(3);

            set.add(4);

            set.add(5);

            new WriteThread("Writer", set).start();

            new ReadThread("Reader", set).start();

      }

}

class WriteThread extends Thread {

      private Set< Integer > set;

      public WriteThread(String name, Set< Integer > set) {

            this.set = set;

            super.setName(name);

      }

      public void run() {

            int count = 6;

            while (true) {

                  try {

                        Thread.sleep(5000);

                  } catch (InterruptedException ex) {

                        ex.printStackTrace();

                  }

                  set.add(count++);

                  System.out.println(super.getName() + " done");

            }

      }

}

class ReadThread extends Thread {

      private Set< Integer > set;

      public ReadThread(String name, Set< Integer > set) {

            this.set = set;

            super.setName(name);

      }

      public void run() {

            while (true) {

                  String output = "\n" + super.getName() + ":";

                  Iterator< Integer > iterator = set.iterator();

                  while (iterator.hasNext()) {

                        Integer next = iterator.next();

                        output += " " + next;

                        // fake processing time

                        try {

                              Thread.sleep(10);

                        } catch (InterruptedException ex) {

                              ex.printStackTrace();

                        }

                  }

                  System.out.println(output);

            }

      }

}

You can see that the read operations outnumber the write ones.

Run this program and observe the result. You will see that the reader thread constantly prints out the elements in the list, whereas the writer thread slowly adds a new number to the list. This program runs forever until you press Ctrl + C to stop it.

Try to change the list implementation from CopyOnWriteArraySet to HashSet like this:

Set< Integer > set = new HashSet< >();

Recompile and run the program again, you will see that the reader thread throws ConcurrentModificationException as soon as the writer thread adds a new element to the list. The reader thread die and only the writer thread alive.

That’s all about CopyOnWriteArraySet.

Understanding ConcurrentHashMap

The next concurrent collection in the java.util.concurrent package I’d like to help you get familiar with today is ConcurrentHashMap - a Map implementation like HashMap and Hashtable, with additional support for concurrency features:

- Unlike Hastable or synchronizedMap which locks the entire map exclusively to gain thread-safety feature, ConcurrentHashMap allows concurrent writer and reader threads. That means it allows some threads to modify the map and other threads to read values from the map at the same time, while Hashtable or synchronizedMap allows only one thread to work on the map at a time. More specifically, ConcurrentHashMap allows any number of concurrent reader threads and a limited number of concurrent writer threads, and both reader and writer threads can operate on the map simultaneously.

                + Reader threads perform retrieval operations such as get, containsKey, size, isEmpty, and iterate over keys set of the map.

                + Writer threads perform update operations such as put and remove.

- Iterators returned by ConcurrentHashMap are weakly consistent, meaning that the iterator may not reflect latest update since it was constructed. An iterator should be used by only one thread and no ConcurrentModificationException will be thrown if the map is modified while the iterator is being used.

ConcurrentHashMap is an implementation of ConcurrentMap which is a subtype of the Map interface. A ConcurrentMap defines the following atomic operations:

- **putIfAbsent(K key, V value)**: associates the specified key to the specified value if the key is not already associated with a value. This method is performed atomically, meaning that no other threads can intervene in the middle of checking absence and association.

- **remove(Object key, Object value)**: removes the entry for a key only if currently mapped to some value. This method is performed atomically.

- **replace(K key, V value)**: replaces the entry for a key only if currently mapped to some value. This method is performed atomically.

- **replace(K key, V oldValue, V newValue)**: replaces the entry for a key only if currently mapped to a given value. This method is performed atomically.

Also note that the methods size() and isEmpty() may return an approximation instead of an exact count due to the concurrent nature of the map. ConcurrentHashMap does not allow null key and null value.

ConcurrentHashMap has such advanced concurrent capabilities because it uses a finer-grained locking mechanism. We don’t delve in to the details of the locking algorithm, but understand that the ConcurrentHashMap uses different locks to lock different parts of the map, which enables concurrent reads and updates.

**\* ConcurrentHashMap Example:**

The following example demonstrates how ConcurrentHashMap is used in a multi-threaded context. The program creates two writer threads and 5 reader threads working on a shared instance of a ConcurrentHashMap.

The writer thread randomly modifies the map (put and remove). Here’s the code:

public class WriterThread extends Thread {

      private ConcurrentMap< Integer, String > map;

      private Random random;

      private String name;

      public WriterThread(ConcurrentMap< Integer, String > map,

                                    String threadName, long randomSeed) {

            this.map = map;

            this.random = new Random(randomSeed);

[this.name](http://this.name/) = threadName;

      }

      public void run() {

            while (true) {

                  Integer key = random.nextInt(10);

                  String value = name;

                  if(map.putIfAbsent(key, value) == null) {

                        long time = System.currentTimeMillis();

                        String output = String.format("%d: %s has put [%d => %s]",

                                                                        time, name, key, value);

                        System.out.println(output);

                  }

                  Integer keyToRemove = random.nextInt(10);

                  if (map.remove(keyToRemove, value)) {

                        long time = System.currentTimeMillis();

                        String output = String.format("%d: %s has removed [%d => %s]",

                                                                        time, name, keyToRemove, value);

                        System.out.println(output);

                  }

                  try {

                        Thread.sleep(500);

                  } catch (InterruptedException ex) {

                        ex.printStackTrace();

                  }

            }

      }

}

The reader thread iterates over each key-value pair in the map and prints it out. Here’s the code:

public class ReaderThread extends Thread {

      private ConcurrentHashMap< Integer, String > map;

      private String name;

      public ReaderThread(ConcurrentHashMap< Integer, String > map, String threadName) {

            this.map = map;

[this.name](http://this.name/) = threadName;

      }

      public void run() {

            while (true) {

                  ConcurrentHashMap.KeySetView< Integer, String > keySetView = map.keySet();

                  Iterator< Integer > iterator = keySetView.iterator();

                  long time = System.currentTimeMillis();

                  String output = time + ": " + name + ": ";

                  while (iterator.hasNext()) {

                        Integer key = iterator.next();

                        String value = map.get(key);

                        output += key + "=>" + value + "; ";

                  }

                  System.out.println(output);

                  try {

                        Thread.sleep(300);

                  } catch (InterruptedException ex) {

                        ex.printStackTrace();

                  }

            }

      }

}

And the main program creates and starts 2 writer threads and 5 reader threads to work concurrently on a shared instance of a ConcurrentHashMap. Here’s the code:

public class ConcurrentHashMapExamples {

      public static void main(String[] args) {

            ConcurrentHashMap< Integer, String > map = new ConcurrentHashMap< >();

            new WriterThread(map, "Writer-1", 1).start();

            new WriterThread(map, "Writer-2", 2).start();

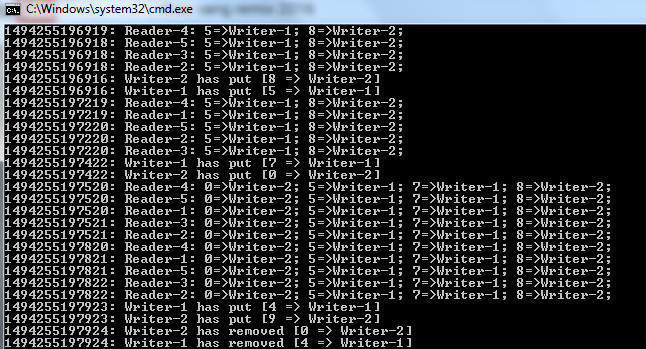
            for (int i = 1; i <= 5; i++) {

                  new ReaderThread(map, "Reader-" + i).start();

            }

      }

}

This program runs forever because all threads run an infinite loop, so you need to press Ctrl + C to stop the program and observe the output. The reader threads let you know that the mp is constantly updated by the writer threads. Here’s a screenshot captured when running the above program on Windows:  
  


**\* Differences between ConcurrentHashMap and HashMap, Hashtable and synchronizedMap:**

HashMap is a non-threadsafe Map which should not be used by multiple threads.

Hashtable is a thread-safe Map that allows only one thread to execute a read/update operation at a time.

synchronizedMap is a thread-safe wrapper on a Map implementation. It is generated by the Collections.synchronizedMap(Map)  factory method. A synchronizedMap also allows only a single thread to work on the map at a time.

And ConcurrentHashMap is a thread-safe Map with greater flexibility and higher scalability as it uses a special locking mechanism that enables multiple threads to read/update the map concurrently.

Therefore, you can use ConcurrentHashMap to replace HashMap/Hastable/synchronizedMap for concurrency needs without locking the whole map.

I hope with this understanding, you will be able to decide when, where and how to use a ConcurrentHashMap.