* 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:

The total balance is reported as 1000 consistently.

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

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:

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 program. 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:

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.

Understanding Synchronization - Using synchronized keyword

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

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:

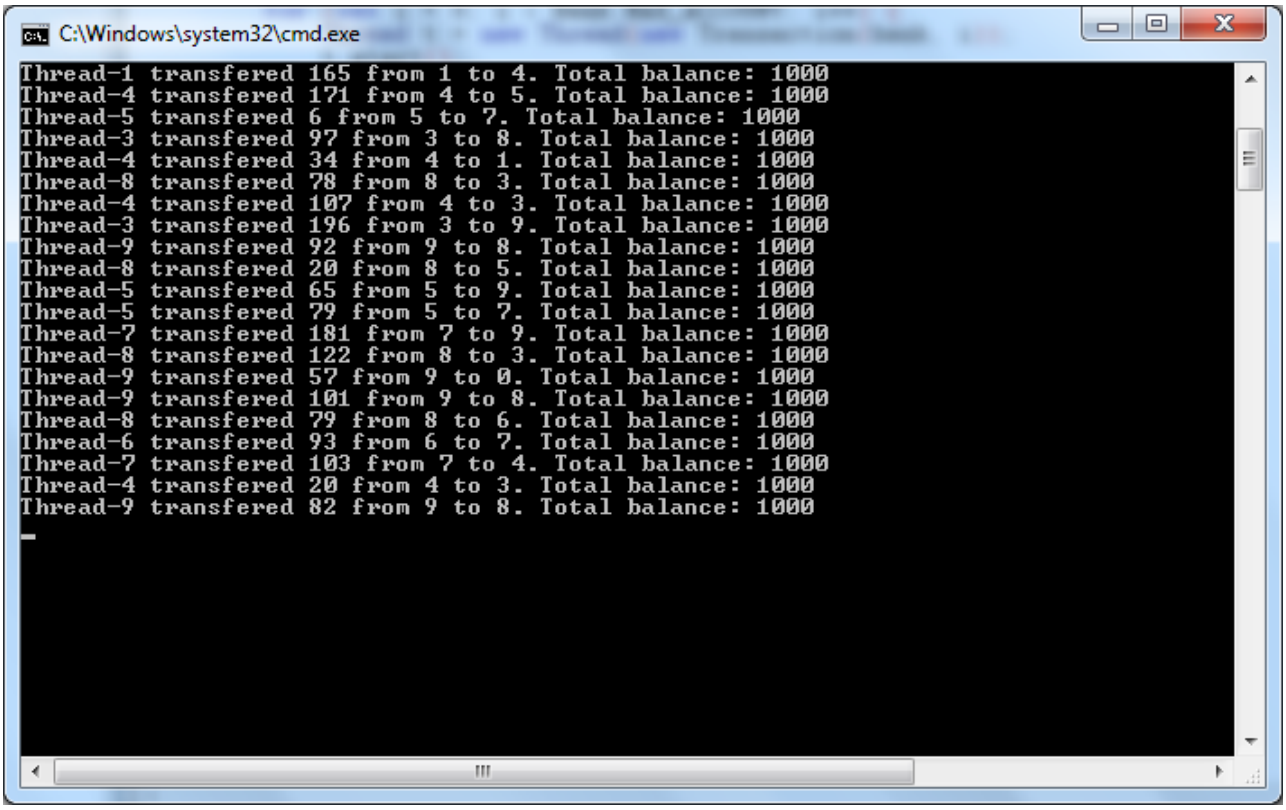
```
public static final int MAX_AMOUNT = 200;
```

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:



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();
        }
  }
```

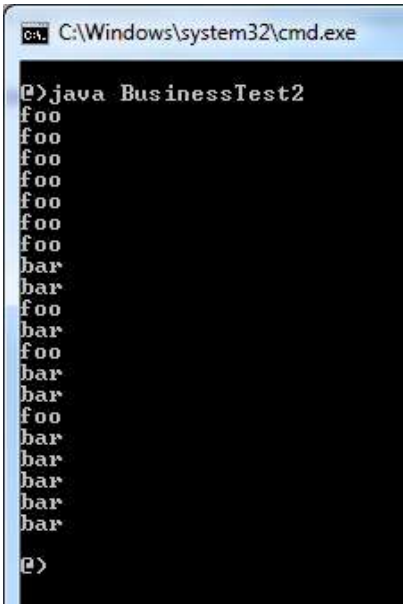
This program creates two threads, one executes the `foo()` method and another executes the `bar()` method on a shared instance of the `Business` class. But deadlock is likely never to occur because one thread can execute and exit a method very quickly so the other thread have chance to acquire the locks.

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

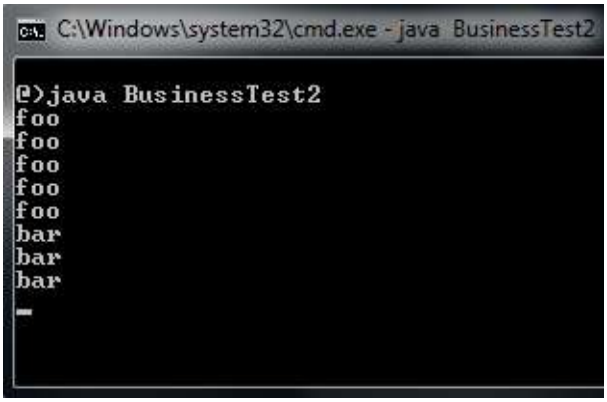
```
  public class BusinessTest2 {
        public static void main(String[] args) {
              Business business = new Business();
              for (int i = 0; i < 10; i++) {
                    new Thread(new Runnable() {
```

```
                        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:



But sometimes it hangs like this:



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();
                  }
```

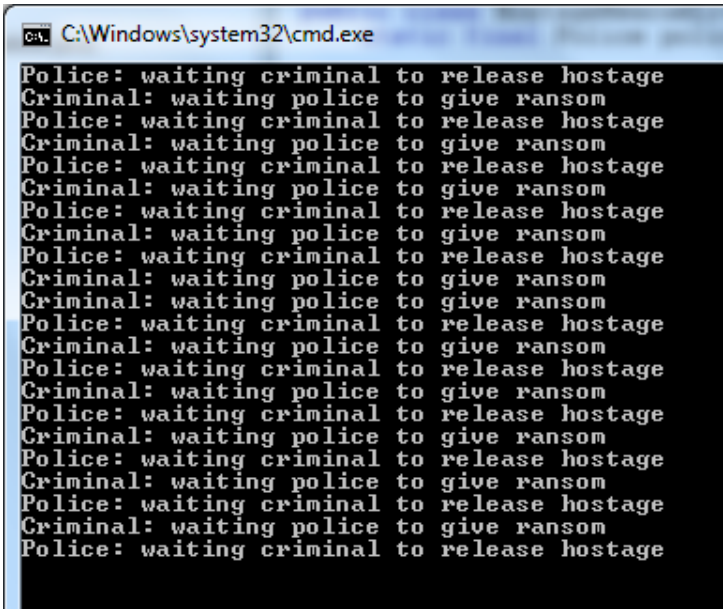
```
            }
            System.out.println("Police: sent money");
            this.moneySent = true;
        }
        public boolean isMoneySent() {
            return this.moneySent;
        }
    }
```

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:



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:

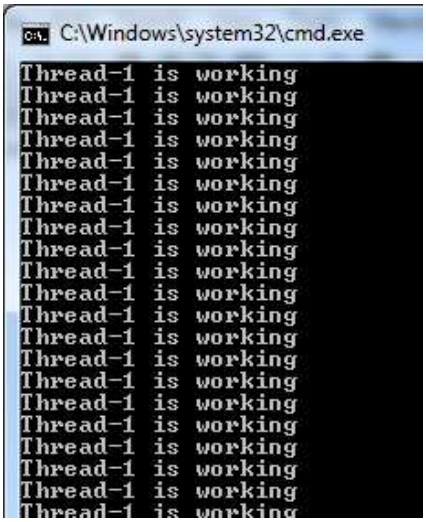
```
is working
```

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();
                                }
                        }).start();
                }
        }
}
```

Compile and run this program and you will see that there's only one thread gets executed:



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.

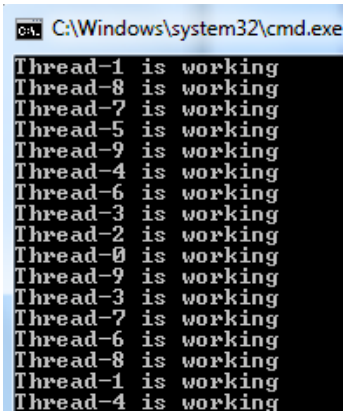
Why does this happen? It's because the while loop runs forever so that the first executed thread never release the lock, causing other threads to wait forever.

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:



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.