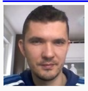
Introduction To Concurrent Programming: A Beginner's Guide



by **Marko Dvečko** - Freelance Software Engineer @ Toptal

#ActorModel #Concurrency #Deadlock #SharedState

What is concurrent programing? Simply described, it's when you are doing more than one thing at the same time.

Not to be confused with parallelism, concurrency is when multiple sequences of operations are run in overlapping periods of time. In the realm of programming, concurrency is a pretty complex subject. Dealing with constructs such as threads and locks and avoiding issues like race conditions and deadlocks can be quite cumbersome, making concurrent programs difficult to write. Through concurrency, programs can be designed as independent processes working together in a specific composition. Such a structure may or may not be made parallel; however, achieving such a structure in your program offers numerous advantages.

In this article, we will take a look at a number of different models of concurrency, how to achieve them in various programming languages designed for concurrency.

Shared Mutable State Model

Let’s look at a simple example with a counter and two threads that increase it. The program shouldn’t be too complicated. We have an object that contains a counter that increases with method increase, and retrieves it with method get and two threads that increase it.

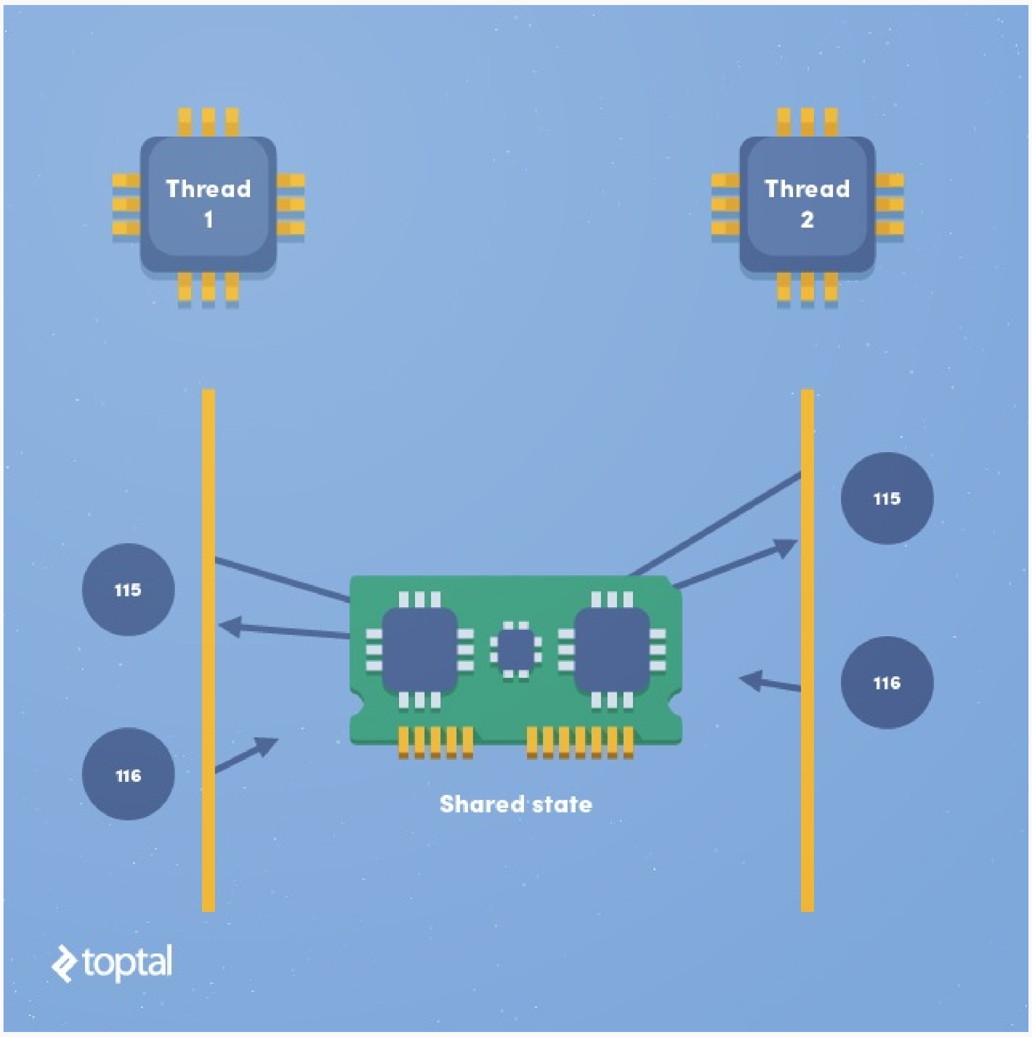
```
//
// Counting.java
//
public class Counting {
    public static void main(String[] args) throws InterruptedException {
        class Counter {
            int counter = 0;
            public void increment() { counter++; }
            public int get() { return counter; }
        }

        final Counter counter = new Counter();

        class CountingThread extends Thread {
            public void  run() {
                for (int x = 0; x < 500000; x++) {
                    counter.increment();
                }
            }
        }

        CountingThread t1 = new CountingThread();
        CountingThread t2 = new CountingThread();
        t1.start(); t2.start();
        t1.join(); t2.join();
        System.out.println(counter.get());
    }
}
```

This naive program is not as naive as it seems at first glance. When I run this program more times I get different results. There are three values after three executions on my laptop.

```
java Counting
553706
java Counting
547818
java Counting
613014
```

What is the reason for this unpredictable behavior? The program increases the counter in one place, in method increase that uses command counter++. If we look at the command byte code we would see that it consists of several parts:

1. Read counter value from memory
2. Increase value locally
3. Store counter value in memory



Now we can imagine what can go wrong in this sequence. If we have two threads that independently increase the counter then we could have this scenario:

1. Counter value is 115
2. First thread reads the value of the counter from the memory (115)
3. First thread increases the local counter value (116)
4. Second thread reads the value of the counter from the memory (115)
5. Second thread increases the local counter value (116)
6. Second thread saves the local counter value to the memory (116)
7. First thread saves the local counter value to the memory (116)
8. Value of the counter is 116

In this scenario, two threads are intertwined so that the counter value is increased by 1, but the counter value should be increased by 2 because each thread increases it by 1. Different threads intertwining influences the result of the program. The reason of the program's unpredictability is that the program has no control of the thread intertwining but operating system. Every time the program is executed, threads can intertwine differently. In this way we introduced accidental unpredictability (non-determinism) to the program.

To fix this accidental unpredictability (non-determinism), the program must have control of the thread intertwining. When one thread is in the method increase another thread must not be in the same method until the first comes out of it. In that way we serialize access to the method increase.

```
//
// CountingFixed.java
//
public class CountingFixed {
    public static main(String[] args) throws InterruptedException {
        class Counter {
            int counter = 0;
            public synchronized void increase() { counter++; }
            public synchronized int get() { return counter; }
        }
        final Counter counter = new Counter();

        class CountingThread extends Thread {
            public void run() {
                for (int i = 0; i < 500000; i++) {
                    counter.increment();
                }
            }
        }

        CountingThread thread1 = new CountingThread();
        CountingThread thread2 = new CountingThread();
        thread1.start(); thread2.start();
        thread1.join(); thread2.join();
        System.out.println(counter.get());
    }
}
```

Another solution is to use a counter which can increase atomically, meaning operation can not be separated into multiple operations. In this way, we don't need to have blocks of code that need to synchronize. Java has atomic data types in java.util.concurrent.atomic namespace, and we'll use AtomicInteger.

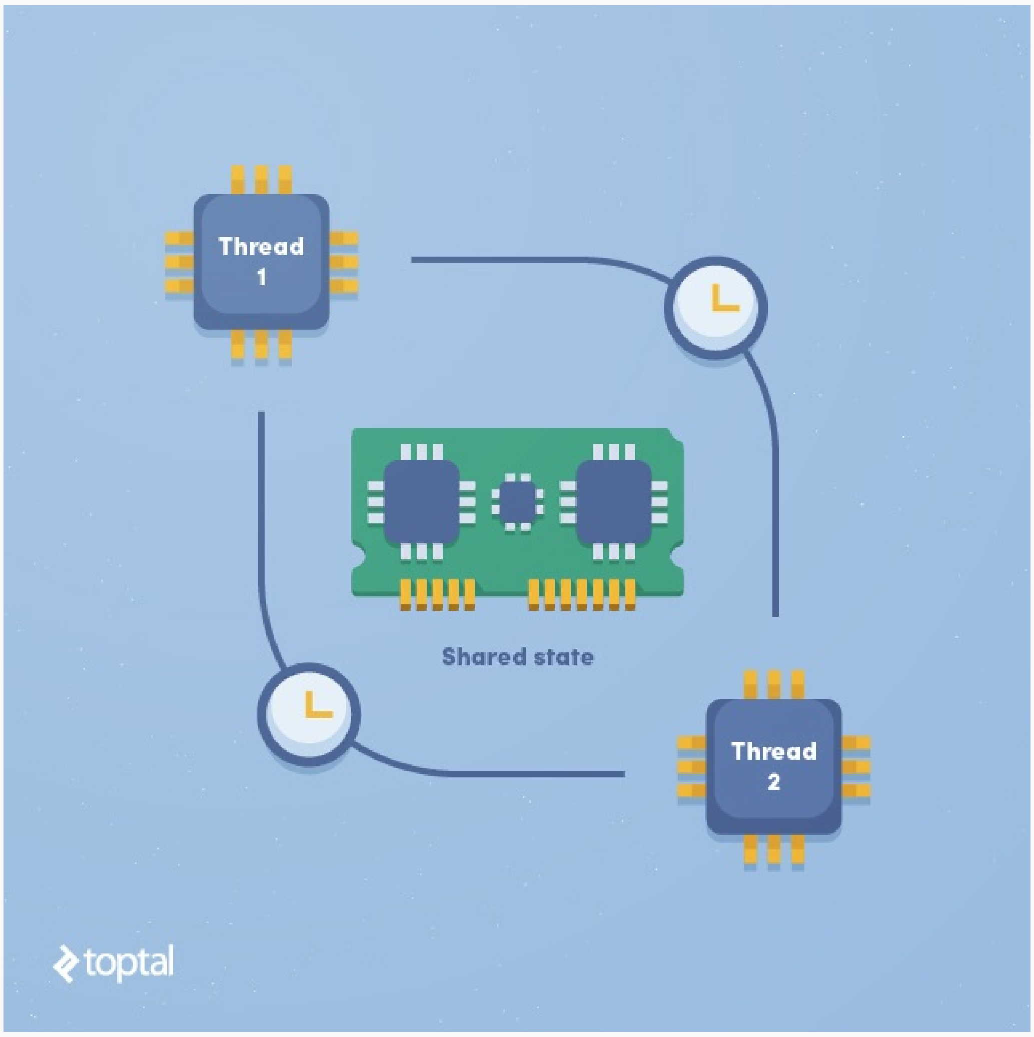
```
//
// CountingBetter.java
//
import java.util.concurrent.atomic.AtomicInteger;

class CountingBetter {
    public static void main(String[] args) throws InterruptedException {
        final AtomicInteger counter = new AtomicInteger(0);

        class CountingThread extends Thread {
            public viod run() {
                for (int i = 0; i < 500000; i++) {
                    counter.incrementAndGet();
                }
            }
        }
        CountingThread thread1 = new CountingThread();
        CountingThread thread2 = new CoutningThread();
        thread1.start(); thread2.start();
        thread1.join(); thread2.join();
        System.out.println(counter.get());
    }
}
```

Atomic integer has the operations that we need, so we can use it instead of the Counter class. It is interesting to note that all methods of atomicinteger do not use locking, so that there is no possibility of deadlocks, which facilitates the design of the program.

Using synchronized keywords to synchronize critical methods should resolve all problems, right? Let's imagine that we have two accounts that can deposit, withdraw and transfer to another account. What happens if at the same time we want to transfer money from one account to another and vice versa? Let's look at an example.

```
//
// Deadlock.java
//
public class Deadlock {
    public static void main(String[] args) throws InterruptedException {
        class Account {
            int balance = 100;
            public Account(int balance) { this.balance = balance; }
            public synchronized void deposit(int amount) { balance += amount; }
            public synchronized boolean withdraw(int amount) {
                if (balance >= amount) {
                    balance -= amount;
                    return true;
                }
                return false;
            }
            public synchronized boolean transfer(Account destination, int amount) {
                if (balance >= amount) {
                    balance -= amount;
                    synchronized(destination) {
                        destination.balance += amount;
                    };
                    return true;
                }
                return false;
            }
            public int getBalance() { return balance; }
        }

        final Account bob = new Account(200000);
        final Account joe = new Account(300000);

        class FirstTransfer extends Thread {
            public void run() {
                for (int i = 0; i < 100000; i++) {
                    bob.transfer(joe, 2);
                }
            }
        }
        class SecondTransfer extends Thread {
            public void run() {
                for (int i = 0; i < 100000; i++) {
                    joe.transfer(bob, 1);
                }
            }
        }

        FirstTransfer thread1 = new FirstTransfer();
        SecondTransfer thread2 = new SecondTransfer();
        thread1.start(); thread2.start();
        thread1.join(); thread2.join();
        System.out.println("Bob's balance: " + bob.getBalance());
        System.out.println("Joe's balance: " + joe.getBalance());
    }
}
```

When I run this program on my laptop it usually gets stuck. Why does this happen? If we look closely, we can see that when we transfer money we are entering into the transfer method that is synchronized and locks access to all synchronized methods on the source account, and then locks destination account which locks access to all synchronized methods on it.



Imagine the following scenario:

1. First thread calls transfer on Bob’s account to Joe’s account
2. Second thread calls transfer on Joe’s account to Bob’s account

3. Second thread decreases amount from Joe's account
4. Second thread goes to deposit amount to Bob's account but waits for first thread to complete transfer.
5. First thread decreases amount from Bob's account
6. First thread goes to deposit amount to Joe's account but waits for second thread to complete transfer.

In this scenario, one thread is waiting for another thread to finish transfer and vice versa. They are stuck with each other and the program cannot continue. This is called deadlock. To avoid deadlock it is necessary to lock accounts in the same order. To fix the program we'll give each account a unique number so that we can lock accounts in the same order when transferring the money.

```
//
// DeadlockFixed.java
//
import java.util.concurrent.atomic.AtomicInteger;

public class DeadlockFixed {
    public static void main(String[] args) throws InterruptedException {
        final AtomicInteger counter = new AtomicInteger(0);
        class Account {
            int balance = 100;
            int order;
            public Account(int balance) {
                this.balance = balance;
                this.order = counter.getAndIncrement();
            }
            public synchronized void deposit(int amount) { balance += amount; }
            public synchronized boolean withdraw(int amount) {
                if (balance >= amount) {
                    balance -= amount;
                    return true;
                }
                return false;
            }
            public boolean transfer(Account destination, int amount) {
                Account first;
                Account second;
                if (this.order < destination.order) {
                    first = this;
                    second = destination;
                }
                else {
                    first = destination;
                    second = this;
                }
                synchronized(first) {
                    synchronized(second) {
                        if (balance >= amount) {
                            balance -= amount;
                            destination.balance += amount;
                            return true;
                        }
                        return false;
                    }
                }
            }
            public synchronized int getBalance() { return balance; }
        }

        final Account bob = new Account(200000);
        final Account joe = new Account(300000);

        class FirstTransfer extends Thread {
            public void run() {
                for (int i = 0; i < 100000; i++) {
                    bob.transfer(joe, 2);
                }
            }
        }
        class SecondTransfer extends Thread {
            public void run() {
                for (int i = 0; i < 100000; i++) {
                    joe.transfer(bob, 1);
                }
            }
        }

        FirstTransfer thread1 = new FirstTransfer();
        SecondTransfer thread2 = new SecondTransfer();
        thread1.start(); thread2.start();
        thread1.join(); thread2.join();
        System.out.println("Bob's balance: " + bob.getBalance());
        System.out.println("Joe's balance: " + joe.getBalance());
    }
}
```

Due to the unpredictability of such mistakes, they sometimes happen, but not always and they are difficult to reproduce. If the program behaves unpredictably, it is usually caused by concurrency which introduces accidental non-determinism. To avoid accidental non-determinism we should in advance design program to take into account all intertwinings.

An example of a program that has an accidental non-determinism.

```
//
// NonDeteminism.java
//
public class NonDeterminism {
    public static void main(String[] args) throws InterruptedException {
        class Container {
            public String value = "Empty";
        }
        final Container container = new Container();

        class FastThread extends Thread {
            public void run() {
                container.value = "Fast";
            }
        }

        class SlowThread extends Thread {
            public void run() {
                try {
                    Thread.sleep(50);
                }
                catch(Exception e) {}
                container.value = "Slow";
            }
        }

        FastThread fast = new FastThread();
        SlowThread slow = new SlowThread();
        fast.start(); slow.start();
        fast.join(); slow.join();
        System.out.println(container.value);
    }
}
```

This program has accidental non-determinism in it. The last value entered in the container will be displayed.

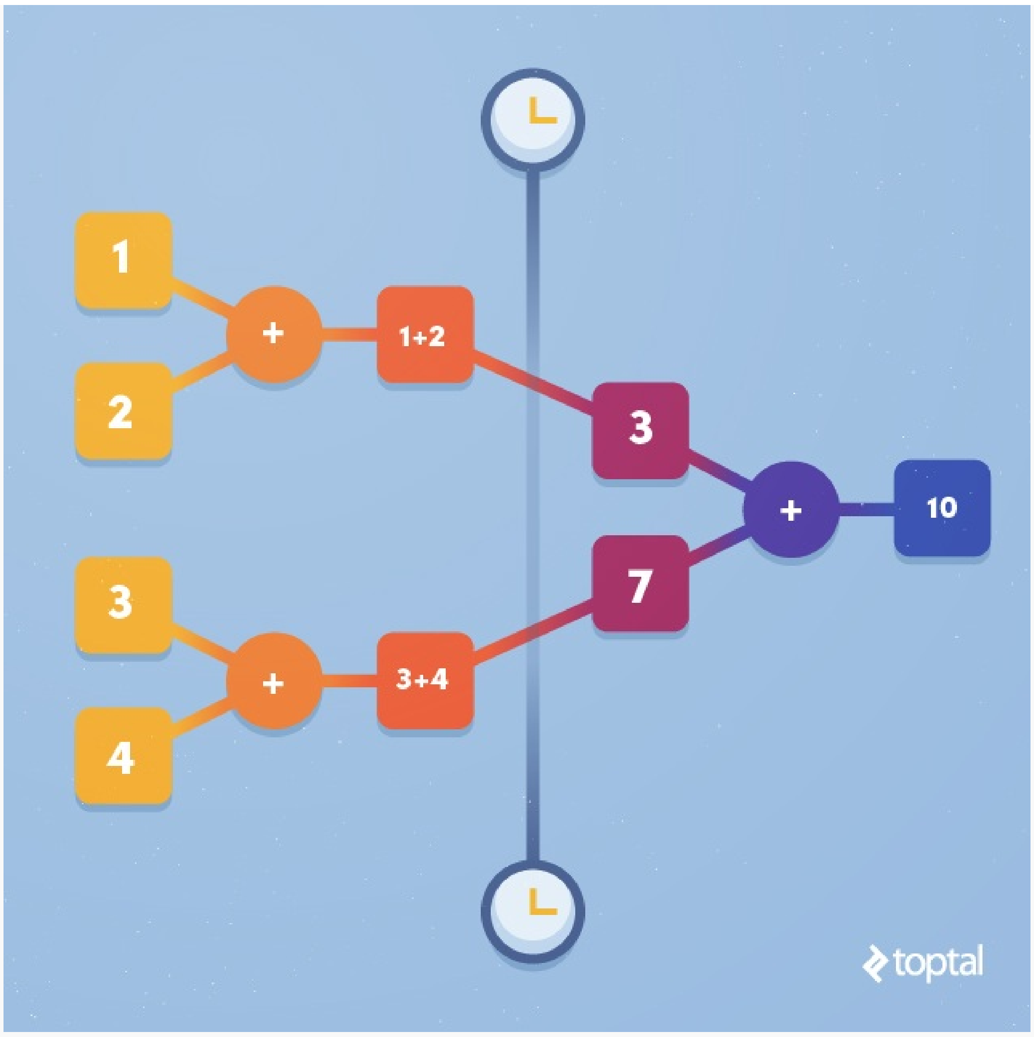
```
java NonDeterminism
Slow
```

Slower threads will enter the value later, and this value will be printed (Slow). But this needs not be the case. What if the computer simultaneously executes another program that needs a lot of CPU resources? We have no guarantee that it will be the slower thread that enters value last because it is controlled by operating system, not the program. We can have situations where the program works on one computer and on the other behaves differently. Such errors are difficult to find and they cause headaches for developers. For all these reasons this concurrency model is very difficult to do right.

Functional Way

Parallelism

Let’s look at another model that functional languages are using. For example we will use Clojure, that can be interpreted using the tool Leiningen. Clojure is a very interesting language with good support for concurrency. The previous concurrency model was with shared mutable state. Classes that we use can also have a hidden state that mutates that we don’t know about, because it is not evident from their API. As we have seen, this model can cause accidental non-determinism and deadlocks if we are not careful. Functional languages have data types that don’t mutate so it can be safely shared without the risk that they will change. Functions have properties as well as other data types. Functions can be created during program execution and passed as parameter to another function or return as a result of the function call.



Basic primitives for concurrent programing are future and promise. Future executes a block of code in another thread and returns an object for the future value that will be entered when the block gets executed.

```
;
; future.clj
;
(let [a (future
          (println "Started A")
          (Thread/sleep 1000)
          (println "Finished A")
          (+ 1 2))
      b (future
          (println "Started B")
          (Thread/sleep 2000)
          (println "Finished B")
          (+ 3 4))]
  (println "Waiting for futures")
  (+ @a @b))
```

When I execute this script the output is:

```
Started A
Started B
Waiting for futures
Finished A
Finished B
10
```

In this example we have two future blocks that are executed independently. Program only blocks when reading the value from the future object that is not yet available. In our case, awaiting both results of future blocks to be summed. Behavior is predictable (deterministic) and will always give the same result because there is no shared mutable state.

Another primitive that is used for concurrency is a promise. Promise is a container in which one can put a value once. When reading promises, the thread will wait until the value of the promise gets filled.

```
;
; promise.clj
;
(def result (promise))
(future (println "The result is: " @result))
(Thread/sleep 2000)
(deliver result 42)
```

In this example, the future will wait to print the result as long as the promise not to be saved value. After two seconds, in the promise will be stored value 42 to be printed in the future thread. Using promises can lead to deadlock as opposed to the future, so be careful when using promise.

```
;
; promise-deadlock.clj
;
(def promise-result (promise))
(def future-result
  (future
    (println "The result is: " + @promise-result)
    13))
(println "Future result is: " @future-result)
(deliver result 42)
```

In this example, we are using the result of the future and the result of the promise. The order of setting and reading values is that the main thread is waiting for a value from the future thread and future thread is waiting for a value from the main thread. This behavior will be predictable (deterministic) and will be played each time the program executes which makes it easier to find and remove error.

Using the future allows the program to continue with the exercise until it needs the result of the execution of the future. This results in faster program execution. If you have multiple processors with the future, you can make parallel execution of program that have predictable (deterministic) behavior (each time gives the same result). That way we better exploit the power of the computer.

```
;
; fibonacci.clj
;
(defn fibonacci[a]
  (if (<= a 2)
    1
    (+ (fibonacci (- a 1)) (fibonacci (- a 2)))))

(println "Start serial calculation")
(time (println "The result is: " (+ (fibonacci 36) (fibonacci 36))))
(println "Start parallel calculation")

(defn parallel-fibonacci[]
  (def result-1 (future (fibonacci 36)))
  (def result-2 (future (fibonacci 36)))
  (+ @result-1 @result-2))
(time (println "The result is: " (parallel-fibonacci)))
```

In this example you can see how the use of future can make better use of a computer’s speed. We have two Fibonacci numbers that add up. We can see that program calculates the result twice, the first time sequentially in a single thread, and the second time in parallel in two threads. As my laptop has a multicore processor, parallel execution works twice as fast as sequential calculation.

The result of executing this script on my laptop:

```
Start serial calculation
The result is:  29860704
"Elapsed time: 2568.816524 msecs"
Start parallel calculation
The result is:  29860704
"Elapsed time: 1216.991448 msecs"
```

Concurrency

To support concurrency and unpredictability in the Clojure programming language, we must use a data type that is variable so other threads can see the changes. The simplest variable data type is atom. Atom is a container which always has the value that can be replaced by another value. The value can be replaced by entering a new value or by calling a function that takes the old value and returns new value which is more frequently used. It is interesting that atom is implemented without locking and it is safe to use in threads, which means that it is impossible to reach deadlock. Internally, atom uses java.util.concurrent.AtomicReference library. Let’s look at a counter example implemented with atom.

```
;
; atom-counter.clj
;
(def counter (atom 0))
(def attempts (atom 0))

(defn counter-increases[]
  (dotimes [cnt 500000]
    (swap! counter (fn [counter]
                     (swap! attempts inc) ; side effect DO NOT DO THIS
                     (inc counter)))))

(def first-future (future (counter-increases)))
(def second-future (future (counter-increases)))
; Wait for futures to complete
@first-future
@second-future
; Print value of the counter
(println "The counter is: " @counter)
(println "Number of attempts: " @attempts)
```

The result of the script execution on my laptop:

```
The counter is: 1000000
Number of attempts: 1680212
```

In this example we use an atom that contains the value of the counter. The counter increases with (swap! counter inc). Swap function works like this: 1. take the counter value and preserve it 2. for this value calls given function that calculates the new value 3. to save new value, it uses atomic operation that checks whether the old value has changed 3a. if the value has not changed it enters a new value 3b. if the value is changed in the meantime, then go to step 1 We see that the function can be called again if the value is changed in the meantime. The value can only be changed from another thread. Therefore, it is essential that the function which calculates a new value has no side effects so that it does not matter if it gets called more times. One limitation of atom is that it synchronizes changes to one value.

```
;
; atom-acocunts.clj
;
(def bob (atom 200000))
(def joe (atom 300000))
(def inconsistencies (atom 0))

(defn transfer [source destination amount]
  (if (not= (+ @bob @joe) 500000) (swap! inconsistencies inc))
  (swap! source - amount)
  (swap! destination + amount))

(defn first-transfer []
  (dotimes [cnt 100000]
    (transfer bob joe 2)))

(defn second-transfer []
  (dotimes [cnt 100000]
    (transfer joe bob 1)))

(def first-future (future (first-transfer)))
(def second-future (future (second-transfer)))
@first-future
@second-future
(println "Bob has in account: " @bob)
(println "Joe has in account: " @joe)
(println "Inconsistencies while transfer: " @inconsistencies)
```

When I execute this script I get:

```
Bob has in account:  100000
Joe has in account:  400000
Inconsistencies while transfer:  36525
```

In this example we can see how we change more atoms. At one point, inconsistency can happen. The sum of two accounts at some time is not the same. If we have to coordinate changes of multiple values there are two solutions:

1. Place more values in one atom
2. Use references and software transactional memory, as we shall see later

```
;
```

```
; atom-accounts-fixed.clj
;
(def accounts (atom {:bob 200000, :joe 300000}))
(def inconsistencies (atom 0))

(defn transfer [source destination amount]
  (let [deref-accounts @accounts]
    (if (not= (+ (get deref-accounts :bob) (get deref-accounts :joe)) 500000)
      (swap! inconsistencies inc))
    (swap! accounts
           (fn [accs]
             (update (update accs source - amount) destination + amount)))))

(defn first-transfer []
  (dotimes [cnt 100000]
    (transfer :bob :joe 2)))


(defn second-transfer []
  (dotimes [cnt 100000]
    (transfer :joe :bob 1)))

(def first-future (future (first-transfer)))
(def second-future (future (second-transfer)))
@first-future
@second-future
(println "Bob has in account: " (get @accounts :bob))
(println "Joe has in account: " (get @accounts :joe))
(println "Inconsistencies while transfer: " @inconsistencies)
```

When I run this script on my computer I get:

```
Bob has in account:  100000
Joe has in account:  400000
Inconsistencies while transfer:  0
```

In the example, coordination has been resolved so that we put more value using a map. When we transfer money from the account, we change all acounts at the time so that it will never happen that the sum of money is not the same.

The next variable data type is agent. Agent behaves like an atom only in that the function that changes the value is executed in a different thread, so that it takes some time for change to become visible. Therefore, when reading the value of the agent it is necessary to call a function that waits until all functions that change the value of the agent are executed. Unlike atoms function that changes the value is called only once and therefore can have side effects. This type can also synchronize one value and cannot deadlock.

```
;
; agent-counter.clj
;
(def counter (agent 0))
(def attempts (atom 0))

(defn counter-increases[]
  (dotimes [cnt 500000]
    (send counter (fn [counter]
                    (swap! attempts inc)
                    (inc counter)))))

(def first-future (future (counter-increases)))
(def second-future (future (counter-increases)))
; wait for futures to complete
@first-future
@second-future
; wait for counter to be finished with updating
(await counter)
; print the value of the counter
(println "The counter is: " @counter)
(println "Number of attempts: " @attempts)
```

When I run this script on my laptop I get:

```
The counter is:  1000000
Number of attempts: 1000000
```

This example is the same as the implementation of the counter with the atom. Only difference is that here we are waiting for all agent changes to complete before reading the final value using await.

The last variable data type are references. Unlike atoms, references can synchronize changes to multiple values. Each operation on reference should be in a transaction using dosync. This way of changing data is called software transactional memory or abbreviated STM. Let’s look at an example with the money transfer in the accounts.

```
;
;  stm-accounts.clj
;
(def bob (ref 200000))
(def joe (ref 300000))
```

```
(def inconsistencies (atom 0))
(def attempts (atom 0))
(def transfers (agent 0))

(defn transfer [source destination amount]
  (dosync
   (swap! attempts inc) ; side effect DO NOT DO THIS
   (send transfers inc)
   (when (not= (+ @bob @joe) 500000)
     (swap! inconsistencies inc)) ; side effect DO NOT DO THIS
   (alter source - amount)
   (alter destination + amount)))

(defn first-transfer []
  (dotimes [cnt 100000]
    (transfer bob joe 2)))

(defn second-transfer []
  (dotimes [cnt 100000]
    (transfer joe bob 1)))

(def first-future (future (first-transfer)))
(def second-future (future (second-transfer)))
@first-future
@second-future
(await transfers)
(println "Bob has in account: " @bob)
(println "Joe has in account: " @joe)
(println "Inconsistencies while transfer: " @inconsistencies)
(println "Attempts: " @attempts)
(println "Transfers: " @transfers)
```

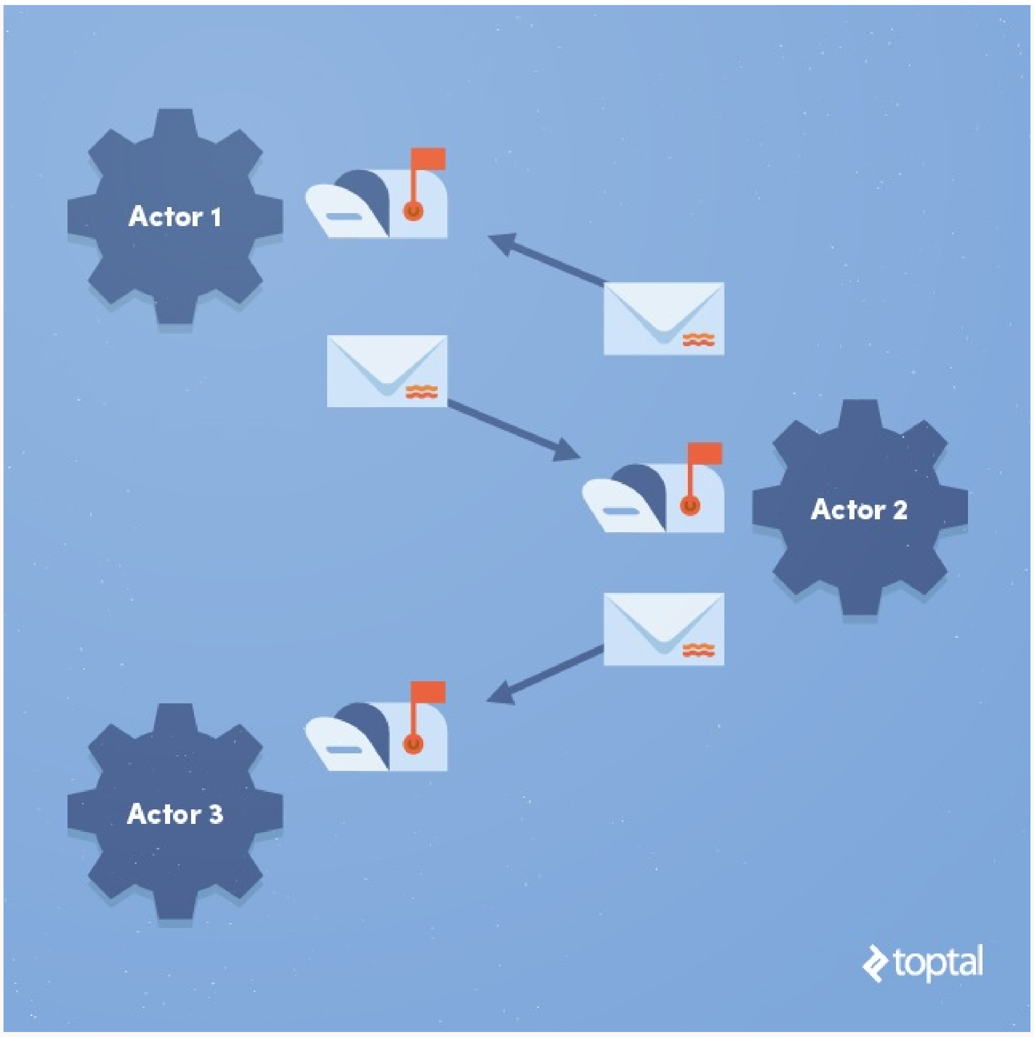
When I run this script, I get:

```
Bob has in account:  100000
Joe has in account:  400000
Inconsistencies while transfer:  0
Attempts:  330841
Transfers:  200000
```

Interestingly, there were more attempts than the number of transactions made. This is because the STM does not use locks, so if there is a conflict, (like two threads trying to change the same value) the transaction will be re-executed. For this reason, the transaction should not have side effects. We can see that the agent which value changes within the transaction behaves predictably. A function that changes the value of the agent will be evaluated as many times as there are transactions. The reason is that the agent is transaction aware. If transaction must have side effects, they should be put into function within the agent. In this way, the program will have predictable behavior. You probably think that you should always use STM, but experienced programmers will often use atoms because atoms are simpler and faster than STM. Of course, that’s if it is possible to make a program in that way. If you have side effects, then there’s no other choice than to use STM and agents.

Actor Model

The following model of concurrency is an actor model. The principle of this model is similar to the real world. If we make a deal to create something with many people, for example a building, then each man at the construction site has their own role. A crowd of people is supervised by the supervisor. If a worker is injured at work, the supervisor will assign the job of the injured man to the others that are available. If necessary he may lead to the site a new man. On the site we have more people who do the work simultaneously (concurrently), but also talking to each other to synchronize. If we put work on the construction site into the program, then every person would be an actor who has a state and executes in its own process, and the talking would be replaced with messages. The popular programming language based on this model is Erlang. This interesting language has immutable data types and functions that have the same properties as other data types. Functions can be created during program execution and passed as arguments to another function or returned as result of function call. I will give examples in the Elixir language that uses the Erlang virtual machine, so I’ll have the same programming model as Erlang just different syntax. The three most important primitives in Elixir are spawn, send and receive. spawn executes function in the new process, send sends the message to the process and receive receives messages that are sent to the current process.



The first example with the actor model will be counter increased concurrently. To make a program with this model, it is necessary to make an actor have the value of the counter and receive message to set and retrieve the value of the counter, and have two actors who will simultaneously increase the value of the counter.

```
#
# Counting.exs
#
defmodule Counting do
  def counter(value) do
    receive do
      {:get, sender} ->
        send sender, {:counter, value}
        counter value
      {:set, new_value} -> counter(new_value)
    end
  end

 def counting(sender, counter, times) do
    if times > 0 do
      send counter, {:get, self}
      receive do
        {:counter, value} -> send counter, {:set, value + 1}
      end
      counting(sender, counter, times - 1)
    else
      send sender, {:done, self}
    end
  end
end

counter = spawn fn -> Counting.counter 0 end

IO.puts "Starting counting processes"
this = self
counting1 = spawn fn ->
  IO.puts "Counting A started"
  Counting.counting this, counter, 500_000
  IO.puts "Counting A finished"
end
counting2 = spawn fn ->
  IO.puts "Counting B started"
  Counting.counting this, counter, 500_000
  IO.puts "Counting B finished"
end

IO.puts "Waiting for counting to be done"
```

```
receive do
  {:done, ^counting1} -> nil
end
receive do
  {:done, ^counting2} -> nil
end

send counter, {:get, self}
receive do
  {:counter, value} -> IO.puts "Counter is: #{value}"
end
```

When I execute this example I get:

```
Starting counting processes
Counting A started
Waiting for counting to be done
Counting B started
Counting A finished
Counting B finished
Counter is: 516827
```

We can see that in the end the counter is 516827 and not 1000000 as we expected. When I ran the script next time, I received 511010. The reason for this behavior is that the counter receives two messages: retrieve the current value and set the new value. To increase the counter, program needs to get the current value, increase it by 1 and set the increased value. Two processes read and write the value of the counter at the same time by using message that are sent to counter process. The order of messages that counter will receive is unpredictable, and the program cannot control it. We can imagine this scenario:

1. Counter value is 115
2. Process A reads the value of the counter (115)
3. Process B reads the value of the counter (115)
4. Process B increases the value locally (116)
5. Process B sets increased value to the counter (116)
6. Process A increases the value of the counter (116)
7. Process A sets increased value to the counter (116)
8. Counter value is 116

If we look at the scenario, two processes increase the counter by 1, and counter gets increased in the end by 1 and not by 2. Such intertwinings can happen an unpredictable number of times and therefore the value of the counter is unpredictable. To prevent this behavior, the increase operation must be done by one message.

```
#
# CountingFixed.exs
#
defmodule Counting do
  def counter(value) do
    receive do
      :increase -> counter(value + 1)

      {:get, sender} ->
        send sender, {:counter, value}
        counter value
    end
  end

 def counting(sender, counter, times) do
    if times > 0 do
      send counter, :increase
      counting(sender, counter, times - 1)
    else
      send sender, {:done, self}
    end
  end
end

counter = spawn fn -> Counting.counter 0 end

IO.puts "Starting counting processes"
this = self
counting1 = spawn fn ->
  IO.puts "Counting A started"
  Counting.counting this, counter, 500_000
  IO.puts "Counting A finished"
end
counting2 = spawn fn ->
  IO.puts "Counting B started"
  Counting.counting this, counter, 500_000
  IO.puts "Counting B finished"
end

IO.puts "Waiting for counting to be done"
receive do
  {:done, ^counting1} -> nil
end
receive do
```

```
  {:done, ^counting2} -> nil
end

send counter, {:get, self}
receive do
  {:counter, value} -> IO.puts "Counter is: #{value}"
end
```

By running this script I get:

```
Starting counting processes
Counting A started
Waiting for counting to be done
Counting B started
Counting A finished
Counting B finished
Counter is: 1000000
```

We can see that the counter has the correct value. The reason for predictable (deterministic) behavior is that the value of the counter increases by one message so that the sequence of messages to increase the counter will not affect its final value. Working with actor model, we have to pay attention to how messages can intertwine and careful design of messages and actions on messages to avoid accidental unpredictability (non-determinism).

How can we transfer money between two accounts with this model?

```
#
# Accounts.exs
#
defmodule Accounts do
  def accounts(state) do
    receive do
      {:transfer, source, destination, amount} ->
        accounts %{state | source => state[source] - amount , destination => state[destination] + amount}
      {:amounts, accounts, sender } ->
        send sender, {:amounts, for account <- accounts do
                        {account, state[account]}
                     end}
        accounts(state)
    end
  end

  def transfer(sender, accounts, source, destination, amount, times, inconsistencies) do
    if times > 0 do
      send accounts, {:amounts, [source, destination], self}
      receive do
        {:amounts, amounts} ->
          if amounts[source] + amounts[destination] != 500_000 do
            Agent.update(inconsistencies, fn value -> value + 1 end)
          end
      end
      send accounts, {:transfer, source, destination, amount}
      transfer(sender, accounts, source, destination, amount, times - 1, inconsistencies)
    else
      send sender, {:done, self}
    end
  end
end

accounts = spawn fn -> Accounts.accounts(%{bob: 200_000, joe: 300_000 }) end
{:ok, inconsistencies} = Agent.start(fn -> 0 end)
this = self
transfer1 = spawn fn ->
  IO.puts "Transfer A started"
  Accounts.transfer(this, accounts, :bob, :joe, 2, 100_000, inconsistencies)
  IO.puts "Transfer A finished"
end
transfer2 = spawn fn ->
  IO.puts "Transfer B started"
  Accounts.transfer(this, accounts, :joe, :bob, 1, 100_000, inconsistencies)
  IO.puts "Transfer B finished"
end

IO.puts "Waiting for transfers to be done"
receive do
  {:done, ^transfer1} -> nil
end
receive do
  {:done, ^transfer2} -> nil
end

send accounts, {:amounts, [:bob, :joe], self}
receive do
  {:amounts, amounts} ->
    IO.puts "Bob has in account: #{amounts[:bob]}"
    IO.puts "Joe has in account: #{amounts[:joe]}"
    IO.puts "Inconsistencies while transfer: #{Agent.get(inconsistencies, fn x -> x end)}"
end
```

When I run this script I get:

```
Waiting for transfers to be done
Transfer A started
Transfer B started
Transfer B finished
Transfer A finished
Bob has in account: 100000
Joe has in account: 400000
Inconsistencies while transfer: 0
```

We can see that money transfer works without inconsistencies, because we have chosen the message transfer to transfer money and message amounts to get the value of accounts which gives us predictable behavior of the program. Whenever we do a transfer of money, the total amount of money at any time should be the same.

Actor model can cause lock and thus deadlock, so use caution when designing the program. The following script shows how you can simulate the lock and deadlock scenario.

```
#
# Deadlock.exs
#
defmodule Lock do
  def loop(state) do
    receive do
      {:lock, sender} ->
        case state do
          [] ->
            send sender, :locked
            loop([sender])
          _ ->
              loop(state ++ [sender])
        end
      {:unlock, sender} ->
        case state do
          [] ->
            loop(state)
          [^sender | []] ->
            loop([])
          [^sender | [next | tail]] ->
            send next, :locked
            loop([next | tail])
          _ ->
            loop(state)
        end
    end
  end

  def lock(pid) do
    send pid, {:lock, self}
    receive do
      :locked -> nil # This will block until we receive message
    end
  end

  def unlock(pid) do
    send pid, {:unlock, self}
  end

  def locking(first, second, times) do
    if times > 0 do
      lock(first)
      lock(second)
      unlock(second)
      unlock(first)
      locking(first, second, times - 1)
    end
  end
end

a_lock = spawn fn -> Lock.loop([]) end
b_lock = spawn fn -> Lock.loop([]) end

this = self
IO.puts "Locking A, B started"
spawn fn ->
  Lock.locking(a_lock, b_lock, 1_000)
  IO.puts "Locking A, B finished"
  send this, :done
end
IO.puts "Locking B, A started"
spawn fn ->
  Lock.locking(b_lock, a_lock, 1_000)
  IO.puts "Locking B, A finished"
  send this, :done
end

IO.puts "Waiting for locking to be done"
receive do
  :done -> nil
```

```
end
receive do
  :done -> nil
End
```

When I run this script on my laptop I get:

```
Locking A, B started
Locking B, A started
Waiting for locking to be done
```

From the output we can see that the processes that lock A and B are stuck. This happens because the first process waits for the second process to release B while second process waiting first process to release A. They are waiting for each other and are stuck forever. To avoid this locking, order should always be the same, or design a program so that it doesn't use lock (meaning that it doesn't wait for a specific message). The following listing always locks first A then B.

```
#
# Deadlock fixed
#
defmodule Lock do
  def loop(state) do
    receive do
      {:lock, sender} ->
        case state do
          [] ->
            send sender, :locked
            loop([sender])
          _ ->
              loop(state ++ [sender])
        end
      {:unlock, sender} ->
        case state do
          [] ->
            loop(state)
          [^sender | []] ->
            loop([])
          [^sender | [next | tail]] ->
            send next, :locked
            loop([next | tail])
          _ ->
            loop(state)
        end
    end
  end

  def lock(pid) do
    send pid, {:lock, self}
    receive do
      :locked -> nil # This will block until we receive message
    end
  end

  def unlock(pid) do
    send pid, {:unlock, self}
  end

  def locking(first, second, times) do
    if times > 0 do
      lock(first)
      lock(second)
      unlock(second)
      unlock(first)
      locking(first, second, times - 1)
    end
  end
end

a_lock = spawn fn -> Lock.loop([]) end
b_lock = spawn fn -> Lock.loop([]) end

this = self
IO.puts "Locking A, B started"
spawn fn ->
  Lock.locking(a_lock, b_lock, 1_000)
  IO.puts "Locking A, B finished"
  send this, :done
end
IO.puts "Locking A, B started"
spawn fn ->
  Lock.locking(a_lock, b_lock, 1_000)
  IO.puts "Locking A, B finished"
  send this, :done
end

IO.puts "Waiting for locking to be done"
receive do
  :done -> nil
end
```

```
receive do
  :done -> nil
End
```

When I run this script on my laptop I get:

```
Locking A, B started
Locking A, B started
Waiting for locking to be done
Locking A, B finished
Locking A, B finished
```

And now, there is no longer a deadlock.

Wrap up

As an introduction to concurrent programming, we have covered a few concurrency models. We haven’t covered all models, as this article would be too big. Just to name a few, channels and reactive streams are some of the other popularly used concurrency models. Channels and reactive streams have many similarities with the actor model. All of them transmit messages, but many threads can receive messages from one channel, and reactive streams transmit messages in one direction to form directed graph that receive messages from one end and send messages from the other end as a result of the processing.

Shared mutable state models can easily go wrong if we don’t think ahead. It has problems of race condition and deadlock. If we have a choice between different concurrent programming models, it would be easier to implement and maintain but otherwise we have to be very careful what we do.

The functional way is a lot easier to reason about and implement. It cannot have deadlock. This model may have worse performance than shared mutable state model, but a program that works is always faster than one that does not work.

Actor model is a good choice for concurrent programming. Although there are problems of race condition and deadlock, they can happen less than in shared mutable state model since the only way for processes to communicate is via messages. With good message design between processes, that can be avoided. If a problem occurs it is then in the order or meaning of messages in communication between the processes and you know where to look.

I hope this article has given you some insight to what concurrent programming is and how it gives structure to the programs you write.

Related: Ruby Concurrency and Parallelism: A Practical Tutorial

About the author

View full profile »
Hire the Author
Marko Dvečko, Croatia
member since November 6, 2015
SalesForce.com APISalesforce.com Data LoaderSalesforce.comSalesforce.com Administrator Certification+3 more
Marko has been a software developer for 12 years. His main focus is in the Salesforce.com platform, and his main interests are in math and functional programming. He holds four Salesforce.com certificates. He started as a C/C++ developer for Windows applications, and then switched to embedded devices. He's spent the last six years working on enterprise software in Java and Salesforce.com APEX and VisualForce. [click to continue...]

Trending articles
Introduction To Concurrent Programming: A Beginner's Guideabout 12 hours agoProgramming Visually With Node-RED: Wiring Up The Internet Of Things With Ease3 days agoIntroduction To BEM Methodology5 days agoEncouraging Action And Timeliness In A Remote Workplace10 days agoGulp Under the Hood: Building a Stream-based Task Automation Tool15 days agoYou Need a Hero: The Project Manager17 days agoClean Code and The Art of Exception Handling18 days agoHSA For Developers: Heterogeneous Computing For The Masses22 days ago
Relevant technologies

- Software
- Java
- Elixir
- Clojure

Toptal Authors

Oguz Gelal
Freelance Software Engineer
Dario Bertini
Freelance Software Engineer
Rohit Boggarapu
Software Engineer @ Adobe
Michał Mikołajczyk
Freelance Software Engineer
Altaibayar Tseveenbayar
Freelance Software Engineer
Elder Santos
Software Engineer
View all authors
About the author

Marko Dvečko
APEX Code Developer
Marko has been a software developer for 12 years. His main focus is in the Salesforce.com platform, and his main interests are in math and functional programming. He holds four Salesforce.com certificates. He started as a C/C++ developer for Windows applications, and then switched to embedded devices. He's spent the last six years working on enterprise software in Java and Salesforce.com APEX and VisualForce.

Toptal Developers

- Android Developers
- AngularJS Developers
- API Developers
- C# Developers
- Django Developers
- Freelance Developers
- Front-End Developers
- Full Stack Developers
- HTML5 Developers
- iOS Developers
- Java Developers

- JavaScript Developers
- jQuery Developers
- .NET Developers
- Node.js Developers
- Objective-C Developers
- OpenGL Developers
- PHP Developers
- Python Developers
- React.js Developers
- Ruby Developers
- Ruby on Rails Developers
- Software Developers
- Unity or Unity3D Developers

See more freelance developers

Join the Toptal community.

Hire a developer
or
Apply as a Developer

Highest In-Demand Talent

- iOS Developer
- Java Developer
- .NET Developer
- Front-End Developer
- UX Designer
- UI Designer

About

- Top 3%
- Clients
- Freelance developers
- Freelance designers
- About Us

Contact

- Contact Us
- Press Center
- Careers
- FAQ

Social

- Facebook
- Twitter
- Google+
- LinkedIn

Toptal

Hire the top 3% of freelance talent