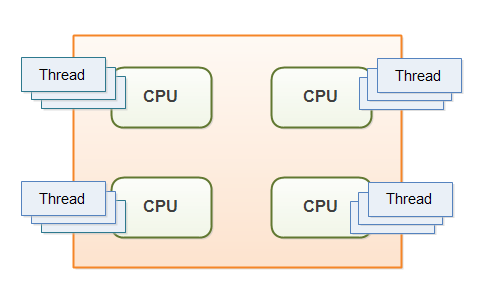
**Java Concurrency** is a term that covers **multithreading, concurrency and parallelism** on the Java platform.

Back in the old days a computer had a single CPU, and was only capable of executing a single program at a time. Later came multitasking which meant that computers could execute multiple programs (AKA tasks or processes) at the same time. It wasn't really "at the same time" though. The single CPU was shared between the programs.

Multithreading can be a great way to increase the performance of some types of programs. However, mulithreading is even more challenging than multitasking. The threads are executing within the same program and are hence reading and writing the same memory simultaneously. This can result in errors not seen in a singlethreaded program. Some of these errors may not be seen on single CPU machines, because two threads never really execute "simultaneously". Modern computers, though, come with multi core CPUs, and even with multiple CPUs too. This means that separate threads can be executed by separate cores or CPUs simultaneously.



If a thread reads a memory location while another thread writes to it, what value will the first thread end up reading? The old value? The value written by the second thread? Or a value that is a mix between the two? Or, if two threads are writing to the same memory location simultaneously, what value will be left when they are done? The value written by the first thread? The value written by the second thread? Or a mix of the two values written?

Without proper precautions any of these outcomes are possible. The behavior would not even be predictable. The outcome could change from time to time. Therefore it is important as a developer to know how to take the right precautions - meaning learning to control how threads access shared resources like memory, files, databases etc. That is one of the topics this Java concurrency tutorial addresses.

**Multithreading Benefits**

The reason multithreading is still used in spite of its challenges is that multithreading can have several benefits. Some of these benefits are:

1. **Better resource utilization:** Imagine an application that reads and processes files from the local file system. Lets say that reading a file from disk takes 5 seconds and processing it takes 2 seconds. Processing two files then takes

5 seconds reading file A

2 seconds processing file A

5 seconds reading file B

2 seconds processing file B

-----------------------

14 seconds total

When reading the file from disk most of the CPU time is spent waiting for the disk to read the data. The CPU is pretty much idle during that time. It could be doing something else. By changing the order of the operations, the CPU could be better utilized. Look at this ordering:

5 seconds reading file A

5 seconds reading file B + 2 seconds processing file A

2 seconds processing file B

-----------------------

12 seconds total

The CPU waits for the first file to be read. Then it starts the read of the second file. While the second file is being read, the CPU processes the first file. Remember, while waiting for the file to be read from disk, the CPU is mostly idle.

1. **Simpler program design in some situations:**  If you were to program the above ordering of reading and processing by hand in a singlethreaded application, you would have to keep track of both the read and processing state of each file. Instead you can start two threads that each just reads and processes a single file. Each of these threads will be blocked while waiting for the disk to read its file. While waiting, other threads can use the CPU to process the parts of the file they have already read. The result is, that the disk is kept busy at all times, reading from various files into memory. This results in a better utilization of both the disk and the CPU. It is also easier to program, since each thread only has to keep track of a single file.
2. **More responsive programs:** Another common goal for turning a singlethreaded application into a multithreaded application is to achieve a more responsive application. Imagine a server application that listens on some port for incoming requests. when a request is received, it handles the request and then goes back to listening. The server loop is sketched below:

while(server is active){

listen for request

process request

}

If the request takes a long time to process, no new clients can send requests to the server for that duration. Only while the server is listening can requests be received.

An alternate design would be for the listening thread to pass the request to a worker thread, and return to listening immediately. The worker thread will process the request and send a reply to the client. This design is sketched below:

while(server is active){

listen for request

hand request to worker thread

}

This way the server thread will be back at listening sooner. Thus more clients can send requests to the server. The server has become more responsive.

**Multithreading Costs:** Don't just multithread-enable an application just because you can. You should have a good idea that the benefits gained by doing so, are larger than the costs. When in doubt, try measuring the performance or responsiveness of the application, instead of just guessing.

1. **More Complex design :** Code executed by multiple threads accessing shared data need special attention. Errors arising from incorrect thread synchronization can be very hard to detect, reproduce and fix.

* When a CPU switches from executing one thread to executing another, the CPU needs to save the local data, program pointer etc. of the current thread, and load the local data, program pointer etc. of the next thread to execute. This switch is called a "context switch".

1. **Context Switching overhead :** Saving the context of one process and loading the context of another process is called Context switching.

* Minimum process required for context switching = 2 exception : for round robin case minimum process required for context switching = 1
* Context switching time is considered as the overhead for the system

1. **Increase Resource consumption** : If number of threads are increased ,more memory is required to save their stack information.

**Concurrency Model:**

A *concurrency model* specifies how threads in the system collaborate to complete the jobs they are given. Different concurrency models split the jobs in different ways, and the threads may communicate and collaborate in different ways.

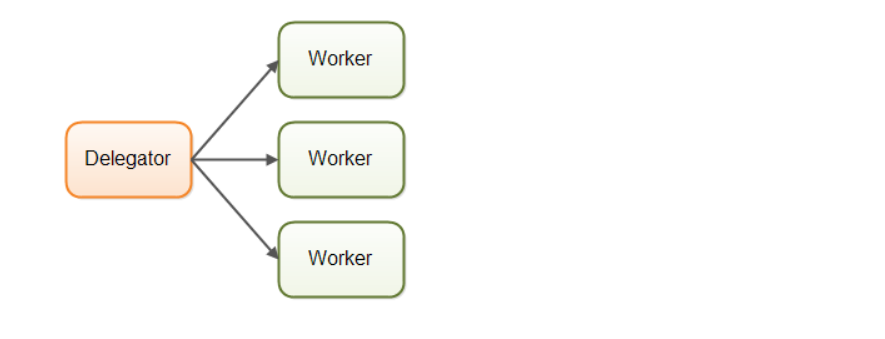
**Concurrency model and Distributed System similarities :** In a concurrent system different threads communicate with each other. In a distributed system different processes communicate with each other (possibly on different computers). Threads and processes are quite similar to each other in nature. That is why the different concurrency models often look similar to different distributed system architectures.

Of course distributed systems have the extra challenge that the network may fail, or a remote computer or process is down etc. But a concurrent system running on a big server may experience similar problems if a CPU fails, a network card fails, a disk fails etc. The probability of failure may be lower, but it can theoretically still happen.

Because concurrency models are similar to distributed system architectures, they can often borrow ideas from each other. For instance, models for distributing work among workers (threads) are often similar to models of [load balancing in distributed systems](http://tutorials.jenkov.com/software-architecture/load-balancing.html).

**Parallel Workers:**

The first concurrency model is what I call the *parallel worker* model. Incoming jobs are assigned to different workers. Here is a diagram illustrating the parallel worker concurrency model:



In the parallel worker concurrency model a delegator distributes the incoming jobs to different workers. Each worker completes the full job. The workers work in parallel, running in different threads, and possibly on different CPUs.

If the parallel worker model was implemented in a car factory, each car would be produced by one worker. The worker would get the specification of the car to build, and would build everything from start to end.

* The parallel worker concurrency model is the most commonly used concurrency model

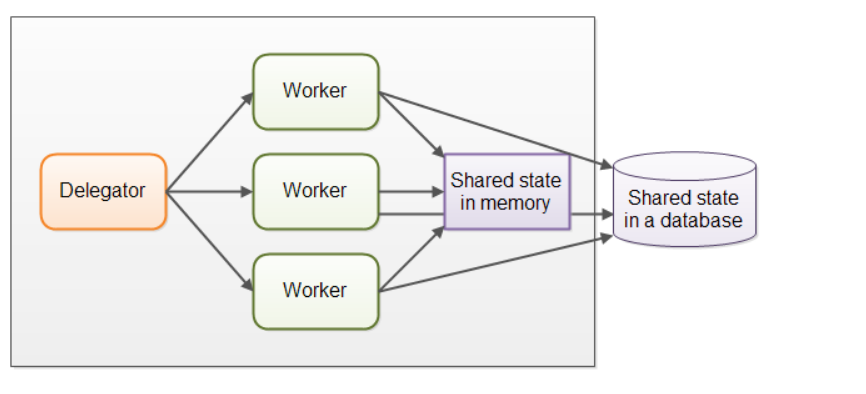
**Advantage of Parallel Workers Model** :

it is easy to understand. To increase the parallelization of the application you just add more workers.

**Disadvantage of Parallel Workers Model** :

### Shared State Can Get Complex:

In reality the parallel worker concurrency model is a bit more complex than illustrated above. The shared workers often need access to some kind of shared data, either in memory or in a shared database. The following diagram shows how this complicates the parallel worker concurrency model:



As soon as shared state sneaks into the parallel worker concurrency model it starts getting complicated. The threads need to access the shared data in a way that makes sure that changes by one thread are visible to the others (pushed to main memory and not just stuck in the CPU cache of the CPU executing the thread). Threads need to avoid [**race conditions**](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html), [**deadlock**](http://tutorials.jenkov.com/java-concurrency/deadlock.html) and many other shared state concurrency problems.

Additionally, part of the parallelization is lost when threads are waiting for each other when accessing the shared data structures.

Modern [**non-blocking concurrency algorithms**](http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html) may decrease contention and increase performance, but non-blocking algorithms are hard to implement.

Persistent data structures are another alternative. A persistent data structure always preserves the previous version of itself when modified. Thus, if multiple threads point to the same persistent data structure and one thread modifies it, the modifying thread gets a reference to the new structure. All other threads keep a reference to the old structure which is still unchanged and thus consistent. The Scala programming contains several persistent data structures.

### Stateless Workers :

Shared state can be modified by other threads in the system. Therefore workers must re-read the state every time it needs it, to make sure it is working on the latest copy. This is true no matter whether the shared state is kept in memory or in an external database. A worker that does not keep state internally (but re-reads it every time it is needed) is called *stateless* .

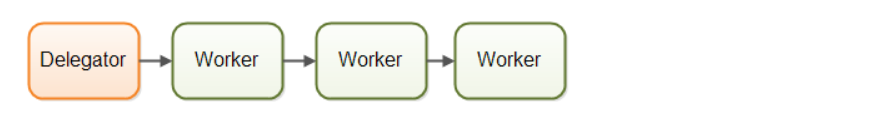
Re-reading data every time you need it can get slow. Especially if the state is stored in an external database.

### Job Ordering is Nondeterministic :

Another disadvantage of the parallel worker model is that the job execution order is nondeterministic. There is no way to guarantee which jobs are executed first or last. Job A may be given to a worker before job B, yet job B may be executed before job A.

## Assembly Line concurrency model :

In this model The workers are organized like workers at an assembly line in a factory. Each worker only performs a part of the full job. When that part is finished the worker forwards the job to the next worker.

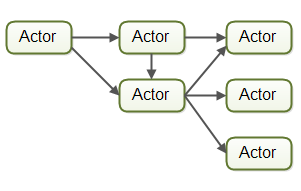


Each worker is running in its own thread, and shares no state with other workers. This is also sometimes referred to as a *shared nothing* concurrency model.

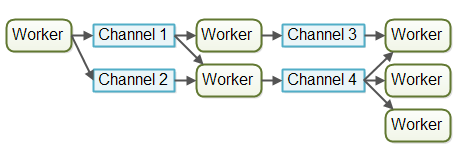
### Actors vs. Channels

Actors and channels are two similar examples of assembly line models.

In the actor model each worker is called an *actor*. Actors can send messages directly to each other. Messages are sent and processed asynchronously.



In the channel model, workers do not communicate directly with each other. Instead they publish their messages (events) on different channels. Other workers can then listen for messages on these channels without the sender knowing who is listening.



At the time of writing, the channel model seems more flexible to me. A worker does not need to know about what workers will process the job later in the assembly line. It just needs to know what channel to forward the job to (or send the message to etc.). Listeners on channels can subscribe and unsubscribe without affecting the workers writing to the channels. This allows for a somewhat looser coupling between workers.

## Assembly Line Advantages:

### No Shared State

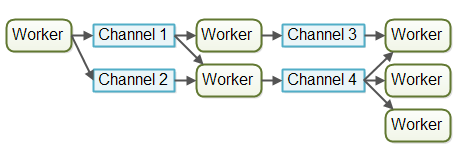
The fact that workers share no state with other workers means that they can be implemented without having to think about all the concurrency problems that may arise from concurrent access to shared state. This makes it much easier to implement workers. You implement a worker as if it was the only thread performing that work - essentially a singlethreaded implementation.

### Stateful Workers

Since workers know that no other threads modify their data, the workers can be stateful. By stateful I mean that they can keep the data they need to operate in memory, only writing changes back the eventual external storage systems. A stateful worker can therefore often be faster than a stateless worker.

**Job ordering is possible :**

It is possible to implement a concurrent system according to the assembly line concurrency model in a way that guarantees job ordering. Job ordering makes it much easier to reason about the state of a system at any given point in time. Furthermore, you could write all incoming jobs to a log. This log could then be used to rebuild the state of the system from scratch in case any part of the system fails. The jobs are written to the log in a certain order, and this order becomes the guaranteed job order. Here is how such a design could look:



Implementing a guaranteed job order is not necessarily easy, but it is often possible. If you can, it greatly simplifies tasks like backup, restoring data, replicating data etc. as this can all be done via the log file(s).

## Assembly Line Disadvantages

The main disadvantage of the assembly line concurrency model is that the execution of a job is often spread out over multiple workers, and thus over multiple classes in your project. Thus it becomes harder to see exactly what code is being executed for a given job.

**Functional Parallelism concurrency model :** The basic idea of functional parallelism is that you implement your program using function calls. Functions can be seen as "agents" or "actors" that send messages to each other, just like in the assembly line concurrency model (AKA reactive or event driven systems). When one function calls another, that is similar to sending a message.

All parameters passed to the function are copied, so no entity outside the receiving function can manipulate the data.

With Java 7 we got the java.util.concurrent package contains the [**ForkAndJoinPool**](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html) which can help you implement something similar to functional parallelism. With Java 8 we got parallel [**streams**](http://tutorials.jenkov.com/java-collections/streams.html) which can help you parallelize the iteration of large collections.

## Which Concurrency Model is Best?

As is often the case, the answer is that it depends on what your system is supposed to do. If your jobs are naturally parallel, independent and with no shared state necessary, you might be able to implement your system using the parallel worker model.

Many jobs are not naturally parallel and independent though. For these kinds of systems I believe the assembly line concurrency model has more advantages than disadvantages, and more advantages than the parallel worker model.

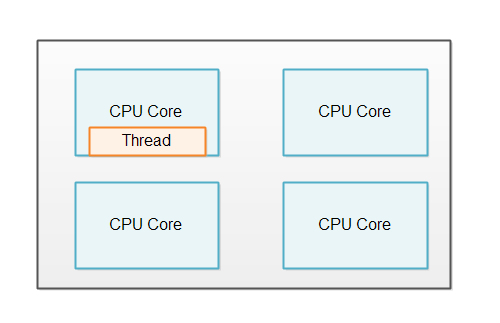
# Same-threading :

# Same-threading is a concurrency model where a single-threaded systems are scaled out to N single-threaded systems. The result is N single-threaded systems running in parallel. A same-threaded system is not a pure single-threaded system, because it contains of multiple threads. But - each of the threads run like a single-threaded system.

# Why Single-threaded System ?

Single-threaded systems have gained popularity because their concurrency models are much simpler than multi-threaded systems. Single-threaded systems do not share any data with other threads. This enables single thread to use non-concurrent data structures, and utilize the CPU and CPU caches better.

Unfortunately, single-threaded systems do not fully utilize modern CPUs. A modern CPU often comes with 2, 4 or more cores. Each core functions as an individual CPU. A single-threaded system can only utilize one of the cores, as illustrated here:



In order to utilize all the cores in the CPU, a single-threaded system can be scaled out to utilize the whole computer.

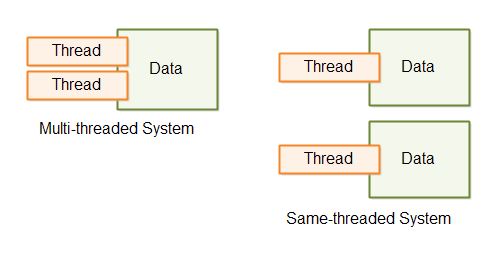
One Thread Per CPU : If a computer contains 4 CPUs, or a CPU with 4 cores, then it would be normal to run 4 instances of the same-threaded system (4 single-threaded systems). The illustration below shows this principle:

### A same-threaded system running on a 4 core CPU.

## No Shared State :

A same-threaded system looks similar to a multi-threaded system, since a same-threaded system has multiple threads running inside it. But there is a subtle difference.

The difference between a same-threaded and a multi-threaded system is that the threads in a same-threaded system do not share state. There is no shared memory which the threads access concurrently. No concurrent data structures etc. via which the threads share data. This difference is illustrated here:



### Single-threaded Microservices

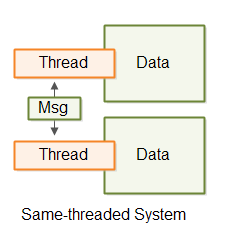
If your system consists of multiple microservices, each microservice can run in single-threaded mode. When you deploy multiple single-threaded microservices to the same machine, each microservice can run a single thread on a sigle CPU.

Microservices do not share any data by nature, so microservices is a good use case for a same-threaded system.

## Thread Communication

If the threads in a same-threaded need to communicate, they do so by message passing. A thread that wants to send a message to thread A can do so by generating a message (a byte sequence). Thread B can then copy that message (byte sequence) and read it. By copying the message thread B makes sure that thread A cannot modify the message while thread B reads it. Once it is copied it is immutable for thread A.

Thread communication via messaging is illustrated here:

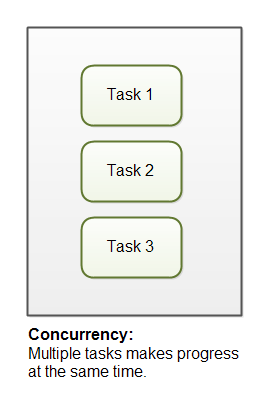


The thread communication can take place via queues, pipes, unix sockets, TCP sockets etc. Whatever fits your system.

**Concurrency vs Parallelism :**

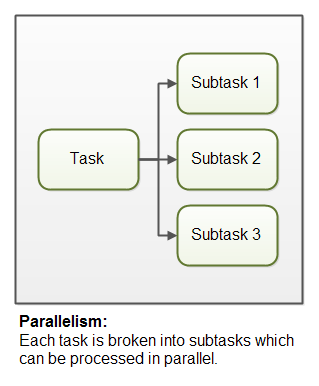
**Concurrency :** Concurrency means that an application is making progress on more than one task at the same time.

if the computer has only one CPU the application may not make progress on more than one task at *exactly the same time*, but more than one task is being processed at a time inside the application. It does not completely finish one task before it begins the next.



## Parallelism

Parallelism means that an application splits its tasks up into smaller subtasks which can be processed in parallel, for instance on multiple CPUs at the exact same time.



**Concurrency vs Parallelism :**

concurrency is related to how an application handles multiple tasks it works on. An application may process one task at a time (sequentially) or work on multiple tasks at the same time (concurrently).

Parallelism on the other hand, is related to how an application handles each individual task. An application may process the task serially from start to end, or split the task up into subtasks which can be completed in parallel.

As you can see, an application can be concurrent, but not parallel. This means that it processes more than one task at the same time, but the tasks are not broken down into subtasks.

An application can also be parallel but not concurrent. This means that the application only works on one task at a time, and this task is broken down into subtasks which can be processed in parallel.

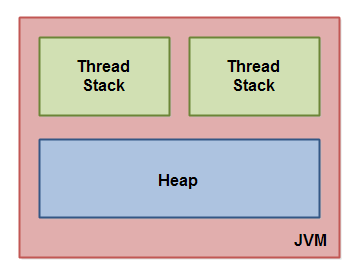
Additionally, an application can be neither concurrent nor parallel. This means that it works on only one task at a time, and the task is never broken down into subtasks for parallel execution.

Finally, an application can also be both concurrent and parallel, in that it both works on multiple tasks at the same time, and also breaks each task down into subtasks for parallel execution. However, some of the benefits of concurrency and parallelism may be lost in this scenario, as the CPUs in the computer are already kept reasonably busy with either concurrency or parallelism alone. Combining it may lead to only a small performance gain or even performance loss. Make sure you analyze and measure before you adopt a concurrent parallel model blindly.

# Java Memory Model : The Java memory model specifies how the Java virtual machine works with the computer's memory (RAM).

## The Internal Java Memory Model

The Java memory model used internally in the JVM divides memory between thread stacks and the heap. This diagram illustrates the Java memory model from a logic perspective:

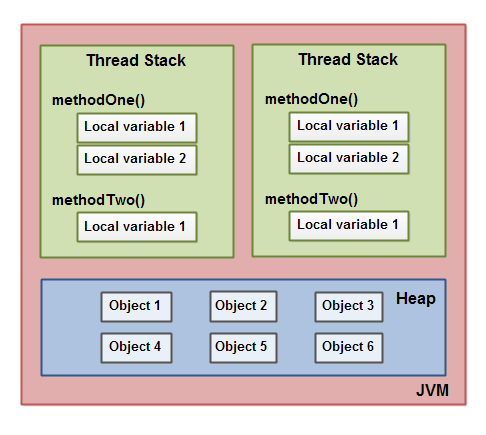


Each thread running in the Java virtual machine has its own thread stack. The thread stack contains information about what methods the thread has called to reach the current point of execution. I will refer to this as the "call stack".The thread stack also contains all local variables for each method being executed .

A thread can only access it's own thread stack. Local variables created by a thread are invisible to all other threads than the thread who created it.

All local variables of primitive types ( boolean, byte, short, char, int, long, float, double) are fully stored on the thread stack and are thus not visible to other threads. One thread may pass a copy of a pritimive variable to another thread, but it cannot share the primitive local variable itself. The heap contains all objects created in your Java application, regardless of what thread created the object.

Conclusion of above discussion :



A local variable may be of a primitive type, in which case it is totally kept on the thread stack.

A local variable may also be a reference to an object. In that case the reference (the local variable) is stored on the thread stack, but the object itself if stored on the heap.

An object may contain methods and these methods may contain local variables. These local variables are also stored on the thread stack, even if the object the method belongs to is stored on the heap.

An object's member variables are stored on the heap along with the object itself. That is true both when the member variable is of a primitive type, and if it is a reference to an object.

Static class variables are also stored on the heap along with the class definition.

Objects on the heap can be accessed by all threads that have a reference to the object. When a thread has access to an object, it can also get access to that object's member variables. If two threads call a method on the same object at the same time, they will both have access to the object's member variables, but each thread will have its own copy of the local variables.

Here is a diagram illustrating the points above:



Two threads have a set of local variables. One of the local variables (Local Variable 2) point to a shared object on the heap (Object 3). The two threads each have a different reference to the same object. Their references are local variables and are thus stored in each thread's thread stack (on each). The two different references point to the same object on the heap, though.

Notice how the shared object (Object 3) has a reference to Object 2 and Object 4 as member variables (illustrated by the arrows from Object 3 to Object 2 and Object 4). Via these member variable references in Object 3 the two threads can access Object 2 and Object 4.

The diagram also shows a local variable which point to two different objects on the heap. In this case the references point to two different objects (Object 1 and Object 5), not the same object. In theory both threads could access both Object 1 and Object 5, if both threads had references to both objects. But in the diagram above each thread only has a reference to one of the two objects.

So, what kind of Java code could lead to the above memory graph? Well, code as simple as the code below:

public class MyRunnable implements Runnable() {

public void run() {

methodOne();

}

public void methodOne() {

int localVariable1 = 45;

MySharedObject localVariable2 =

MySharedObject.sharedInstance;

//... do more with local variables.

methodTwo();

}

public void methodTwo() {

Integer localVariable1 = new Integer(99);

//... do more with local variable.

}

}

public class MySharedObject {

//static variable pointing to instance of MySharedObject

public static final MySharedObject sharedInstance =

new MySharedObject();

//member variables pointing to two objects on the heap

public Integer object2 = new Integer(22);

public Integer object4 = new Integer(44);

public long member1 = 12345;

public long member1 = 67890;

}

If two threads were executing the run() method then the diagram shown earlier would be the outcome. The run() method calls methodOne() and methodOne() calls methodTwo().

methodOne() declares a primitive local variable (localVariable1 of type int) and an local variable which is an object reference (localVariable2).

Each thread executing methodOne() will create its own copy of localVariable1 and localVariable2 on their respective thread stacks. The localVariable1 variables will be completely separated from each other, only living on each thread's thread stack. One thread cannot see what changes another thread makes to its copy of localVariable1.

Each thread executing methodOne() will also create their own copy of localVariable2. However, the two different copies of localVariable2 both end up pointing to the same object on the heap. The code sets localVariable2 to point to an object referenced by a static variable. There is only one copy of a static variable and this copy is stored on the heap. Thus, both of the two copies of localVariable2 end up pointing to the same instance of MySharedObject which the static variable points to. The MySharedObjectinstance is also stored on the heap. It corresponds to Object 3 in the diagram above.

Notice how the MySharedObject class contains two member variables too. The member variables themselves are stored on the heap along with the object. The two member variables point to two other Integer objects. These Integer objects correspond to Object 2 and Object 4 in the diagram above.

Notice also how methodTwo() creates a local variable named localVariable1. This local variable is an object reference to an Integer object. The method sets the localVariable1 reference to point to a new Integerinstance. The localVariable1 reference will be stored in one copy per thread executing methodTwo(). The two Integer objects instantiated will be stored on the heap, but since the method creates a new Integerobject every time the method is executed, two threads executing this method will create separate Integerinstances. The Integer objects created inside methodTwo() correspond to Object 1 and Object 5 in the diagram above.

Notice also the two member variables in the class MySharedObject of type long which is a primitive type. Since these variables are member variables, they are still stored on the heap along with the object. Only local variables are stored on the thread stack.