Chapter 2

Asynchronous Programming in the Java Virtual Machine (JVM)

The Java language is one of the mainstream object oriented programming languages that supports a programming to interfaces discipline [Lea00]. It has evolved into a platform to design and implement applications in several domains of both research and industry [VLFGL01, PS12, NAB$^+$11], along with supporting its community with new language constructs and features. With applications reaching exascale dimensions in terms of data volumes and requiring a lot of computing power, focus has increased towards the development of numerous libraries and frameworks with an attempt to provide distribution and concurrency at the level of Java language [SPCA14, SPCA15]. Java is an object-oriented language, that is based on high-level concepts of interfaces and classes. These classes contain fields and methods to encapsulate state and behavior. At runtime, dynamically generated class instances (objects) are created and interact with each other via method calls. The general behavior of these calls is synchronous, as a caller object will block its execution until it receives a result from the execution of the method's block inside the callee object. A Java program's entry point is the `main` method which creates a single thread with its own stack and context. The basic sequential execution model generates a stack of method calls between objects that run on a single thread.

The Java Virtual Machine(JVM) allows an application to have multiple user and daemon threads of execution running concurrently. All threads have an assigned priority in the JVM that orders the scheduling of threads. When it is created initially, a thread receives the same priority as its parent, so in general application-level threads have the same priority. A thread may also be marked as a daemon before it is started, allowing the JVM to stop when only daemon threads are left running in the application. This daemon property is also inherited from the parent thread, if not explicitly set. When a JVM is started, a single user thread is created and its execution begins with the block in the `main` method defined in a class. From that point the JVM continues to create, start, suspend and stop threads by following the control flow of the program. The JVM is then stopped either when the `System.exit( )` method is explicitly called by the user or all user threads have completed execution, either normally or exceptionally.

Multi-threaded execution is an essential feature of the Java platform. Parallel execution takes place by spawning a thread that runs code asynchronously on the same JVM. The basic approach of asynchronous execution in Java is to extend the `Thread` class and specify the behavior by overriding its `run` method. This method's body is empty by default and will run on a separate thread. Listing 2.1 contains an example of creating a new Thread object. Once the thread is instantiated, the user needs to explicitly call the `start` method to execute the body of `run` in parallel.

In Java, the most straightforward way of communication between threads is via the state of shared variables in the program memory, as there is no direct manner in which a thread can invoke an instruction

Listing 2.1: Explicit Thread Creation

```
class MyThread extends Thread{

 @Override
 public void run( ) {
  //thread control flow
 }
}

//main block or another class
MyThread t = new MyThread( );
t.start( );
```

on another thread. All objects that are instantiated in a program may be accessed and modified by the running threads which hold a reference to them and it is up to the programmer to ensure that concurrent access is synchronized. Accesses and modifications to objects by more than one thread at a time will lead to corrupting data and race conditions.

In Java, class fields can be defined using the `static` keyword making their scope global. Static fields are loaded and instantiated only once when the class is loaded without requiring an object instance of that class. As such they may be accessed from anywhere in the program by existing threads.

When a new thread is spawned it has a separate call stack and context of execution. The call stack is used to save the frames generated by sequences of nested synchronous method calls between objects. The stack size in general limits the depth of these sequences to the point where a `StackOverFlowError` is generated when there is not enough space to allocate a new frame. The size of each stack depends on the JVM and Operating System(OS) and can also be specified by the user through the program's environment variables and virtual machine (VM) arguments. A typical default value for 64bit Operating Systems is 1MB, and can easily reach as large as 2MB as object references now require 8 bytes of space compared to the 4 bytes required on 32bit systems. With a 2MB stack size, a program that runs 500 threads can allocate 1GB of main memory for the threads alone, even though just a small percentage of those threads would require the full 2MB stack to execute its control flow. This makes the basic mechanism for a block of code to be run asynchronously very expensive memory-wise. The problem further increases when an application requires multiple context switches between several threads keeping the suspended threads living in the system. As this number becomes very large, the thread explosion affects the main memory thus the application's performance.

The first part of this chapters covers the concurrency utilities extending the above basic multi-threaded model to facilitate asynchronous programming and communication in Java. The second part of the chapter presents the motivation for extending the research towards the Scala language by illustrating its lightweight and easy-to-use syntax for concurrent programming through two abstractions: actors and futures. The last part of this chapter covers related work on introducing coroutines in JVM based languages and the challenges encountered when to ensure correct functionality together with actors and futures.

2.1 Java Concurrency Utilities

Java, by design, does not have a straightforward way of specifying asynchronous method calls in programming code. As mentioned before a method call implicitly blocks the caller, and unless specified through the basic thread model (see Listing 2.1), the flow of a program is sequential and on a single thread. However, Java does provide interfaces and classes together with constructs that offer more flexible and intuitive ways of coding asynchronous communication, spawning and managing threads. There are also many Java packages that offer the programmer utilities for specifying synchronization points between threads based on state or method return values. This section summarizes the main classes that facilitate thread management and synchronization from the programmer's perspective.

Spawning Threads Java offers two interfaces that define tasks that are to be run asynchronously and allow the user to specify the piece of code to be run on a separate thread. These are the `Runnable` and `Callable` interfaces that offer the `run` and `call` methods respectively, that contain the control flow to be executed in parallel. The difference between the two interfaces is that objects that implement `Callable` execute methods that return a value, while `Runnable` objects have methods that do not a return value. Listing 2.2 shows how an anonymous class that extends the `Runnable` interface can be passed as a parameter to a `Thread` constructor. This offers a much more compact way of specifying a block of code to run asynchronously, but still requires an explicit start of the thread.

Listing 2.2: Thread Creation using Anonymous Classes

```
Thread t = new Thread(new Runnable( ) {
  @Override
   public void run( ) {
    // thread control flow
   }
  });

t.start( );
```

An anonymous class allows instantiation of an object that implements an interface (in this case `Runnable`) without specifying a class name and the behavior of the overridden method (in this case `run`) will only apply to this object. As such, the user bypasses the need to explicitly create a separate class that either implements `Runnable` or extends `Thread`, being especially useful when needing to create small asynchronous tasks. The behavior of the threads is now defined as part of the arguments passed to the `new` construct rather than in a separate class prior to running the task in parallel. This provides programatically more flexibility, the possible parallel behaviors do not need to be defined separately for small functions. Starting with Java 8, instances of `Runnable` and `Callable` can also be created using lambda expressions [Lam]. An example of creating a `Runnable` using a lambda expression is given in Listing 2.3. The code in this listing allows the user to express an instance of a class (like `MyThread`) containing a single method even more compactly. This instance can then be passed as well to a `Thread` constructor or other concurrency utilities to be discussed further in this section.

Listing 2.3: Declaring a Runnable using a lambda expression

```
Runnable task = ( )−>{
 //thread control flow
}
```

Managing Threads Already since Java version 5, there has been support for concurrent programming through its `java.util.concurrent` package and throughout its development the Java platform has also introduced new high-level concurrency Application Programming Interfaces (APIs) to facilitate user experience when developing parallel applications. This package introduces data structures with high-level wrappers to allow creation and management of thread pools to facilitate creating and starting threads. Thread pools allow dynamic control of the number of threads created in the application as well as control over the tasks that are accepted. They also expose methods to specify ordering inside thread pool queues and how tasks are scheduled, suspended and completed. The basic interface that supports this is the `Executor` interface which runs submitted tasks. This decouples task submission from the mechanics of how the submitted task will be scheduled, assigned a thread object and run. An executor simplifies the explicit instantiation and start of a new thread in Listing 2.2 using a single call to the `execute` method as shown in Listing2.4 and since Java 8 this can be simplified further by lambda expressions.

Listing 2.4: Executor Usage

```
Executor e = //initialization of Executor and its properties
e.execute(new Runnable( ) {

  @Override
  public void run( ) {
    // thread control flow
  }
});

//using a lambda expression
e.execute( ( )−>{
  //thread control flow
});
```

The `execute` method has no default implementation and needs to be overridden in order to specify how to run the tasks submitted to it. The method allows the programmer to define certain scheduling policies and approaches. For example, in Listing 2.5 the `Executor` is extended to call the `run` method synchronously inside the body of `execute`. Therefore all tasks submitted to such an executor by one thread will be run sequentially.

Listing 2.5: Synchronous Executor Implementation

```
class SynchronousExecutor implements Executor {

  public void execute(Runnable r) {
    r.run( );
  }
}

```

Another basic approach of extending an executor is such that each task is run in parallel on a separate `Thread` instance as in Listing 2.6. We will see the importance of these two approaches throughout the rest of the thesis.

Listing 2.6: Asynchronous Executor Implementation

```
class AsynchronousExecutor implements Executor {

  public void execute(Runnable r) {
    new Thread(r).start( );
  }
}
```

```

```

Java offers several default implementations of the `Executor` interface that have predefined `execute` methods. The `Executors` class provides factory methods to obtain instances of these implementations. This class is different `Executor` class, as it only provides static factory and utility methods to create predefined configurations for various types of executors, without any specific methods that control thread creation, scheduling and execution. One such implementation is the thread pool which allows the programmer to control the number of threads in the system as well as dynamically adjust them during program execution. In this thesis there are three default implementations that were used throughout the development of the solutions presented in the next chapters:

- `Executors.newCachedThreadPool( )`
- `Executors.newFixedThreadPool(int)`
- `Executors.newForkJoinPool()`

The first one creates a thread pool that instantiates new threads when they are required, but will continue to reuse available idle threads that have been constructed previously. The main advantage of this type of thread pool is that it improves the performance of programs that invoke a lot of small asynchronous tasks. One of the case studies researched in this thesis is specifically tailored to test performance of such a program. If a sequence of short tasks are submitted via the `execute` method, the first few tasks will be assigned newly created threads, while subsequent tasks reuse the same threads once they become idle. The idle threads also have a timeout within which they can be reused by subsequent tasks, otherwise they are terminated and garbage collected. Thus, the thread pool will not consume any resources if it is idle for a long time.

The second construct creates a thread pool with a fixed number of threads passed as a parameter and reuses them for execution. The integer number passed as a parameter represents the maximum number of threads that may be active in the application. All subsequent tasks that are submitted will be placed in a shared unbounded queue until a thread is available. A new thread is created in the pool only if an existing thread is terminated (either successfully or not) and tasks are available for execution in the queue. The main advantage of this thread pool is that it will not starve the system out of resources regardless of how many tasks the program attempts to execute in parallel.

The third construct offers a thread pool which employs a work stealing strategy. The existing threads in the pool have a proactive behaviour attempting to execute existing tasks in the queue if they are idle. This type of pool is especially powerful when executing tasks which in turn spawn subtasks, as the work-stealing mechanism bypasses the process of scheduling tasks and assigning them a thread. Thus it is also useful for processing tasks that perform small computations.

The return type of these three factory methods (and the type of the aforementioned thread pools) is an `ExecutorService` which extends `Executor`. This is a more versatile interface that provides more methods to execute different types of tasks such as instances of `Runnable`, `Callable` or a collection of tasks. This is done through the overloaded `submit` method that can take either one of the three arguments and has a similar functionality to the `execute` method of the parent interface. This allows for more fine-grained control of asynchronous tasks. An example of creating a thread pool and then submitting a task instantiated using lambda expressions is given in Listing 2.7. The task does not necessarily need to be declared separately as in Listing 2.3, especially if its a small computation that is only required once.

Listing 2.7: Using a predefined thread pool to run a task asynchrnously

```
ExecutorService pool = Executors.newCachedThreadPool( );
pool.submit( ( ) −> {
 return 10;
});
```

The `Executor Service` also offers methods to manage termination of the thread pool. An unused Executor Service should be shut down to allow reclamation of its resources. The user can safely shut it down, which will cause it to reject new tasks This can be done using two different methods.

- The `shutdown( )` method will continue to run previously submitted tasks to execute before terminating.
- The `shutdownNow( )` method prevents queued tasks from starting and attempts to stop currently executing tasks.

Synchronizing on Return Values. Java provides a large collection of interfaces and classes that support synchronization of tasks running in parallel including mutexes, semaphores and barriers. Our focus, however, is on the `Future` interface which is a general concurrency abstraction, that encapsulates a method's result to be returned upon completion of a method call that is associated with that future. The interface provides methods to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation. A simple usage of a future is like the one shown in Listing2.8. Here a lambda expression is used to create a `Callable` instance since the method returns a value. The return type of `submit` is a `Future` which is a dynamically generated unique reference to the task to be executed asynchronously. The current thread can then synchronize and obtain the result by calling the `get( )` method. This is a very simple and intuitive synchronization mechanism, yet it has some important implications. The current thread will be blocked until the future completes. This means that the thread will remain live in the main memory occupying the space that was allocated for its context.

Listing 2.8: Using a future to synchronize on the result

```
ExecutorService pool = Executors.newCachedThreadPool( );
Future<Double> f = pool.submit( ( )–> {
            Double x = ... //calculate x
            return x;
           });
// perform other computations in parallel before using the result
Double result = f.get( );
//subsequent code that will only execute when the thread is released
```

The result can only be retrieved when the computation has completed, thus releasing the calling thread, loading its context and executing subsequent code. Cancellation of a task represented by a future may be performed by the `cancel( )` method. The `get( )` can also be used to determine if the task completed normally, was cancelled or has thrown an `Exception`. Once execution has finished, the computation cannot be cancelled. A particular extension of the `Future` interface is the `CompletableFuture`. This implementation provides methods for the developer to create, process and complete a future manually. We want to observe three important methods:

- `complete(T value)` - This method set the future's result to the `value` passed as a parameter, unless the future has already completed.
- `completeExceptionally(Throwable)` - If not already completed, causes invocations of get( ) and related methods to throw the given exception.
- `CompletableFuture.completedFuture(T value)` - This static factory method creates object instances that behave like futures that have completes with the result `value`.

2.2 Scala Abstractions for Concurrent Programming

Scala is an object-oriented programming language that also has support for functional programming. It is intended to compile to Java bytecode such that it can provide language interoperability with Java and allow the usage and integration of libraries designed in either language. With respect to concurrent programming in Scala, this section covers the inner workings of futures and their intended usage, as well as two libraries that support the actor programming model.

Scala Futures In Scala, the package called `scala.concurrent` contains the trait `Future` to provide a way to reason about performing many operations in parallel without blocking until a synchronization point. Traits in Scala are similar to Java interfaces, except they may contain methods which have a default implementation. Unlike Java futures, the entire computation code that is to be done asynchronously can be declared in a `Future` trait like in Listing 2.9.

Listing 2.9: Declaration of a Future in Scala

```
val f: Future[Double] = future {
  val x = ...
  //calculate value of x
  return x;
}
```

To bypass the explicit creation of threads and overwriting of the `run( )` method, Scala futures require an `Execution Context`. Similar to the Java `Executor`, it is responsible for thread creation, execution and termination. The package also provides a global static thread pool which is a default `ExecutionContext` implementation that is backed by a `ForkJoinPool` object suitable for execution of a large number of small asynchronous tasks. To reduce performance penalties caused by a large number of threads, the `ForkJoinPool` object sets a default limit to the number of threads it contains, known as level, to the number of available processors (value yielded by the `Runtime.availableProcessors` function). This parallelism level can be overridden by setting one (or more) of the following VM attributes:

- `scala.concurrent.context.minThreads`
- `scala.concurrent.context.numThreads`
- `scala.concurrent.context.maxThreads`

The parallelism level will be set to `numThreads` as long as it remains within the interval [`minThreads; maxThreads`]. An `ExecutionContext` can also be created from a Java `Executor` using `ExecutionContext.fromExecutor` method. Optionally, developers are allowed to extend the `ExecutionContext` trait to customize their own execution contexts.

In Scala futures make use of callbacks by default, to minimize as much as possible the blocking effect of `get()` operations and therefore the number of blocked threads in the system. Scala provides a general callback method `onComplete` to execute upon completion of a future. The body of the callback has to pattern match on the state of the completed future as shown in Listing 2.10. This state is of abstract type `Try`, which represents a computation whose final result may be a completed value or an exception and therefore can be one of two case classes: `Success` or `Failure` which contain as an argument the result or exception respectively.

Listing 2.10: Callback definition in Scala

```
f onComplete {
  case Success(result)=>process(result);
  case Failure(error)=> println("Computation␣yielded␣error" + error);
}
```

Scala futures also provide more specific methods to define the control sequence to be executed once a future completes, as well as a recovery block if the asynchronous task completed with an exception. These two methods are `onSuccess` (Listing 2.11) and `onFailure` (Listing 2.12) and avoid the need to pattern match on the state of the future.

Listing 2.11: Callback to execute when the future completes sucessfully

```
f onSuccess {
  case result => {
    process(result)
  }
}
```

Listing 2.12: Callback to execute when the future yields an error

```
f onFailure {
  case error => {
    println("Computation␣yielded␣error" + error);
  }
}
```

This is the mechanism specific to Scala through which a result can be used or an error can be handled from a completed future. It is a much more efficient way compared to the approach where the thread that is using the future result has to block using the `get( )` method until it is available. It is important to note that callback function is executed on a separate thread and as such needs to explicitly handle synchronization issues if any of the data structures and operations in the function are not thread safe.

Scala Actors The `scala.actors` package offers an API for concurrent programming with actors. The package provides both asynchronous and synchronous message transmission (the latter are implemented by exchanging several asynchronous messages). Moreover, actors may communicate using futures where requests are handled asynchronously, but return a representation (the future) that allows to await the reply. The typical approach in Scala is that of messages to have any type and thus are only checked at run-time whether the receiver can handle them. All actors contain a mailbox which stores messages until they are processed.

Scala provides a trait hierarchy starting with the basic implementation of actors that can send and receive messages, to supporting more complex capabilities such as obtaining the reference of the sender, replying to a message or grouping actors for monitoring and correct terminations purposes. The API also provides control structures and traits for managing actors in terms of scheduling, error handling and remote communication.

Actor-based programs use the `Actor` interface to define classes like in Listing 2.13. The example shows two actors running a ping-pong benchmark. This is a simple example where two actors (Ping and Pong) exchange a fixed number (count) of empty requests (pings) and replies (pongs) and then the program stops.

Listing 2.13: Creating a Scala Actor

```
class PingActor(count: int, pong: Actor) extends Actor {

}

class PongActor extends Actor {

}
```

To define the message exchange between the two actors, one first needs to define the objects that represent the messages like in Listing 2.14. These are defined as case objects to use Scala pattern matching when defining the behavior for receiving each message.

Listing 2.14: Messages as Case Objects

```
case object PingMessage
case object PongMessage
case object StopMessage
```

The actual running behavior is then defined by overriding the `act` method and using `receive` to define the control flow for each received message. This is shown in Listing 2.15 for the `PingActor` class.

Listing 2.15: Defining Scala Actor Behavior

```
def act( ) {
  var pingsLeft = count − 1
  pong ! PingMessage

  while (true) {
    receive {
      case PongMessage =>

        if (pingsLeft > 0) {
          pong ! PingMessage
          pingsLeft −= 1
        } else {
          pong ! StopMessage
          exit( )
        }
    }
  }
}
```

The Actor model in Scala [HO09] does provide a suspension mechanism, but its use is *not* recommended because it actually blocks the whole thread and causes degradation of performance. It is possible to register a *continuation* piece of code to run upon completion of a future, but that will run in a separate thread which breaks the actor semantics and may cause race conditions inside the actor.

Akka Actor Library Akka actors [Hal12, Gup12] are similar to Scala actors and can be used in both Java and Scala, but are more intuitive and less cumbersome to use in the latter. To define the behavior of an actor in Akka, one needs to extend the `Actor` trait and override the `receive` method. A simple example of defining an Akka Actor is shown in Listing 2.16. Java libraries for programming actors like Akka [SM01] mainly provide pure asynchronous message passing which does not support the use of application programming interfaces (API) because, a message is only typed as a Java Object, so there is no static typing of messages, nor are they part of the actor interface.

An important challenge in both Scala and Akka actors is that messages can only be checked at runtime (they need to be pattern matched in order for the actor to execute a particular control flow). This means that the programmer needs to define for all the possible messages a specific type(see Listing 2.14). Then an actor must pattern match on a given message and execute the matching block of code. This block is often a call to an internal method defined in the actor class. These libraries however do not provide a way to bypass this pattern flow that executes at runtime in favor of a straightforward call to the internal method that is meant to execute asynchronously. The Akka Typed module attempts to solve this challenge

Listing 2.16: Akka Actor

```
import akka.actor.Actor

class MyActor extends Actor {

def receive = {
  case "Hello" => println("HelloWorld")
  case _ => println("received_unknown_message")
  }
}
```

by defining messages as case classes like in Listing 2.17. This provides a close resemblance to methods with arguments, but still requires defining an internal block or method corresponding to the flow to be executed asynchronously.

Listing 2.17: Akka Typed Example

```
object PingActor {
  final case class PingMessage(replyTo: ActorRef[PongMessage])
  final case class PongMessage( from: ActorRef[PingMesssage])
  }
```

2.3 Coroutine Support

Coroutines are programming abstractions of state and control flow in a program that allow for multiple entry points for suspending and resuming execution at certain locations. It is a very powerful concept especially in object-oriented programming for organizing control flow of a large number of small tasks. The current existing support for coroutines in JVM can be categorized by two main approaches: one which operates on source code level and one which operates on the bytecode level.

Main examples of bytecode manipulation are Apache Commons Javaflow [Com] and Kilim [SM08]. Even though bytecode manipulation allows for more flexibility, it has several disadvantages regarding maintainability and portability. Further, the application of debugging techniques becomes more involved and source-code based static analysis tools become unusable.

A straightforward way to support coroutines at source-code level in Java (see [Sch11]) is by allocating a thread to every "routine" that can be suspended, since a thread naturally contains already all the information about the call stack and local variables. However that does not scale because threads are well known to be heavyweight in Java. In Scala, macros are also a viable approach to implementing coroutines. Macros are an experimental feature in Scala that allow a programmer to write code at the level of abstract syntax trees and thus to instruct the compiler to generate code differently.

Scala-coroutines project [Sto] uses this feature to implement low-level coroutine support for Scala with explicit suspension and resume points which however in general are prone to errors. Kotlin [Jan17]supports coroutines natively but actors are not first class citizens. Kotlin actors are implemented as coroutines, which by definition ensures a single thread of execution within the actor. However one cannot process the messages inside actors in a coroutine manner. In other words, it is not possible to process other messages if one message is suspended.

2.4 Summary

Asynchronous calls can be emulated in Java using the basic thread programming pattern or constructs that simplify this pattern. However there is not a straightforward way to issue these calls similar to synchronous calls. A big research challenge in this thesis is to have a mechanism for issuing a call to method (using its signature), executing it on a separate thread and capturing its result in a future. The goal is to have a method call similar to the model of the method call issued in the researched modelling language, ABS. Emulation of the asynchronous call is supported, but it involves declaration of a few classes and instances, possible duplicate code if more than one call is required, or unwanted code that may result from misuse of the generic classes. There is not standalone construct for a method call that type-checks the call to be of the correct signature. In Java, Scala or Akka, the notion of an asynchronous method call is not present as an integrated part of the language. Bridging this gap between the asynchronous call model and its current complex syntax in JVM-based languages and libraries is a significant part of this thesis.