Coroutines in Java

Excerpt from:

Serializable Coroutines for the HotSpot^{TM} Java Virtual Machine [25]

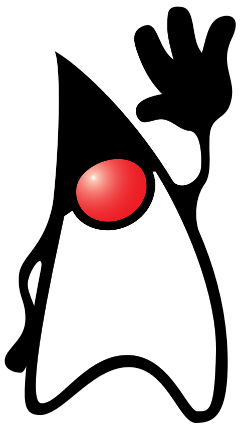
Lukas Stadler
Johannes Kepler University Linz, Austria

September 27, 2011

Contents

Chapter 1

Motivation



This chapter shows why a coroutine implementation in a Java Virtual Machine is useful and who will benefit from it. It also introduces two advanced concepts, namely thread-to-thread migration and serialization/deserialization.

Coroutines are a programming language concept that allows for explicit, cooperative and stateful switching between subroutines. They are a powerful and versatile concept that provides a natural control abstraction for many problems. For example, producer/consumer problems can often be implemented elegantly with coroutines [6].

Within a thread, coroutines do not need to be synchronized because the points of control transfer can be chosen such that no race condition can occur.

Coroutines also provide an easy way to inverse recursive algorithms into iterative ones. For an example of this see Figure 4.6 on page 23.

Coroutines can be simulated by putting the state of a method execution into an object, similar to how closures [15] are usually implemented. In this scheme local variables are stored in heap objects while the task is suspended, so that they can be restored later on. Only coroutines allow the compiler to fully optimize local variables, e.g., put them into processor registers or stack slots. They can thus benefit from the fact that their state is not accessible from the outside, while heap-allocated objects lead to code that is compiled as if such access was possible.

Because of this versatility many new programming languages, such as Ruby [10], Go [12] and Lua [14], incorporate coroutines as essential language or runtime environment features. Language implementations that rely on an underlying virtual machine (VM), such as a Java Virtual Machine (JVM), need to emulate coroutines if they are not available. The most common ways to do so are using synchronized threads and compile-time transformations, both of which have significant drawbacks.

Thus it can be expected that a native coroutine implementation for a production-quality JVM, such as the HotSpot™ Virtual Machine, will provide significant value to language implementation, framework and application developers.

1.1 Advanced Features

Simple coroutines, which are constrained to one thread, are useful for many use cases, but sometimes even more functionality is required:

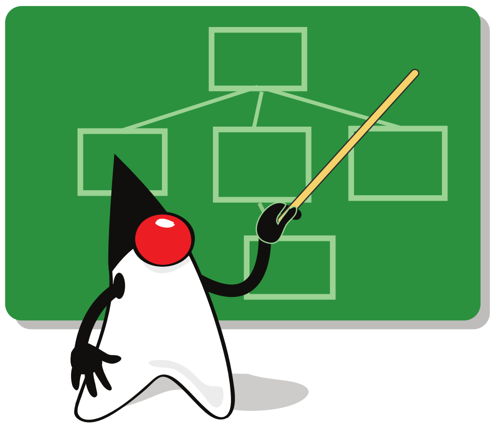
Thread-to-Thread Migration
: For server applications that use coroutines to manage user sessions it might be beneficial to resume a coroutine in a different thread than the one in which it was created.

Serialization/Deserialization
: In other situations it might even be necessary to move a coroutine into permanent storage, like a database. This is useful for use cases like long-lasting transactions, checkpointing or agents that move between VMs.

A coroutine system that includes these concepts provides enough functionality for a large number of use cases.

Chapter 2

Coroutines



This chapter introduces the basic concepts of coroutines, along with a short history of coroutines. It ends with an overview of common coroutine implementation techniques.

Coroutines [18] are a concept has been used since the early days of electronic data processing. The term "Coroutine" was coined in 1963 by Conway [6] as "each module ... may be coded as an autonomous program that communicates with adjacent modules as if they were input and output subroutines". They are present in numerous programming languages, such as Go, Icon, Lua, Perl, Prolog, Ruby, Tcl, Simula, Python, Modula-2, and many others. However, none of the top five languages of the TIOBE programming languages index [27] (Java, C, C++, PHP and Basic) supports coroutines without additional libraries.

From a language design perspective, coroutines are a generalization of subroutines. When they are invoked, they start execution at their first statement. A

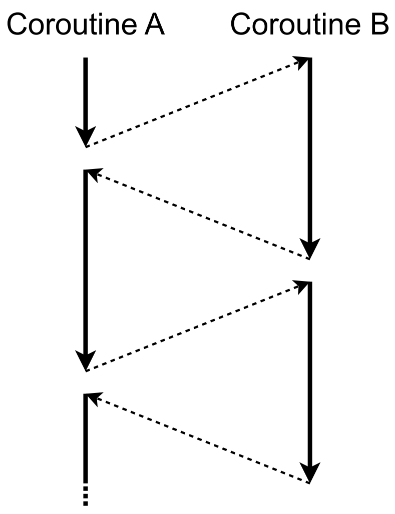


Figure 2.1: Interlocked execution of coroutines

coroutine can transfer control to some other coroutine (typically using a *yield* statement). Coroutines can transfer control back and forth, as shown in Figure 2.1.

The state of the local variables at the transfer point is preserved. When control is transferred back, the coroutine resumes the preserved state and continues to run from the point of the transfer. At its end or at the encounter of a return statement a coroutine dies, i.e., passes control back to its caller or some other coroutine as defined by the semantics of the coroutine system.

Coroutines have often been seen as inferior alternatives to full-blown threads, because of the manual context switching. However, there are situations in which manual context switching makes sense or is even desired (e.g., producer/consumer problems, discrete event simulation, and non-blocking server implementations).

Figure 2.2 shows a simple comparison between subroutine and coroutine execution[1]. The task that the methods in this example should achieve is to return `"Alpher"` the first time the method next() is called and `"Bethe"` the second time. While the subroutine needs to explicitly implement an ad hoc state machine, the coroutine can simple yield a value to the caller without destroying the method's execution frame. Thus, when the coroutine is called the second time it will resume immediately after the **`yield`** statement and return the second value.

[1]Note that this example uses pseudo-code, the actual coroutine API is introduced in Section 4.

```java
// Subroutine
class Container {
  boolean first = false;
  public String next() {
    if (first) {
      first = false;
      return "Alpher";
    } else {
      first = true;
      return "Bethe";
      }
  }
}
```

```java
// Coroutine
class Container {
  public String next() {
    yield "Alpher";
    return "Bethe";
  }
}
```

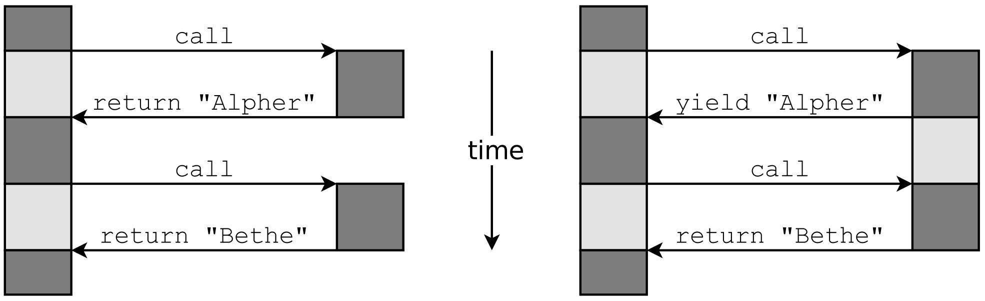


Figure 2.2: Comparison of subroutine (left) and coroutine execution (right)

If a third value were to be returned by the coroutine version, it would simply need to be changed to `yield "Bethe"` and `return "Gamow"`.

2.1 Coroutine Characteristics

The fact that coroutines automatically preserve their execution context is sometimes called *automatic stack management* [2]. Any algorithm that uses coroutines can be re-implemented with state machines, which is considered to be *manual stack management*.

In contrast to other programming language concepts (e.g., continuations), the semantics of coroutines have no strict definition. Coroutines may not even be recognizable as such, and depending on their semantics and expressiveness they are called generators [13, 18], coexpressions [13], fibers [23], iterators [18], green threads [24], greenlets [22], or cooperative threads.

This section presents the most common attributes used to describe coroutine variants.

2.1.1 Stackful vs. Stackless

Coroutines that can only yield from within their main method are called *stackless*. They can be implemented by compile-time transformations, but are only useful for a limited range of applications. An example of this is C#'s implementation of iterator methods using the **`yield`** keyword, which causes the compiler to perform a transformation that turns the method into a state machine that implements the `IEnumberable` interface.

If coroutines can yield from subsequently called methods, they are called *stackful*.

2.1.2 First-class Coroutines

Only coroutines that are not tied to a specific language construct (like for-each loops) and that are programmer-accessible objects are considered to be *first-class* coroutines. Programming languages that limit the usage of coroutines to certain situations usually do so to allow the compiler to transform the coroutine code into ordinary sequential code.

2.1.3 Symmetric vs. Asymmetric Coroutines

Asymmetric coroutines are bound to a specific caller and can only transfer control back to this caller. The coroutine in Figure 2.2 is an asymmetric coroutine. The asymmetry lies in the fact that there are distinct and complementary operations for transferring control to and from a coroutine (**`call`** and **`yield`**/**`return`**). This also means that an asymmetric coroutine is bound to return to its original caller.

Symmetric coroutines, on the other hand, are not bound to return to their predecessor coroutine [18]. There is only one operation, typically also called **`yield`**, which is used to transfer control to another coroutine. Depending on the coroutine implementation the target coroutine can either be specified explicitly (**`yieldTo`**) and/or implicitly (**`yield`**). Using **`yield`**, without specifying a target coroutine, usually involves some kind of scheduler that selects a coroutine to run.

2.2 History

At the assembly language level coroutine techniques have been employed for a long time, although the name "Coroutine" was not used.

The name "Coroutine" was first mentioned in 1963 by Conway [6]. His coroutines were a tool for efficiently implementing multi-pass processes, like compilers. His usage of coroutines is similar to UNIX pipes, where the output of one process is used as the input of another process.

In Simula I and its successor Simula 67 [8], Ole-Johan Dahl and Kristen Nygaard introduced coroutines as a fundamental operation. They had a specific interest in using coroutines for simulation purposes, because the semantics of coroutines are more adequate for simulation than the semantics of subroutines. Thus, Simula I includes a concept similar to coroutines, called *simulation*. Simula 67 introduced the keywords `call`, `detach` and `resume`, which provide a more generic implementation of coroutines. They realized and exploited the fact that coroutines can be used for implementing other language features.

Implementations of Smalltalk [11], which first appeared in 1972, implement the execution stack as a first-class citizen, which allows for a trivial coroutine implementation.

Scheme [1], created in 1975, is one of the first languages that implemented first-class continuations. They are an even more powerful concept than coroutines and can be used as an implementation vehicle for coroutines.

Modula-2 [29], designed and developed between 1977 and 1980 by Niklaus Wirth, also includes coroutines as lightweight, quasi-concurrent processes. They are controlled by the low-level procedures `NEWPROCESS` and `TRANSFER`. The coroutines in Modula-2 are intended primarily for producer/consumer problems, similar to Conway's original design.

The standard C library includes functions named `setjmp` and `longjmp`, which can theoretically be used to implement coroutine-like features, but these rarelyused functions are highly platform dependent and awkward to use. The `setcontext` family of functions, available in POSIX C libraries, is suited for coroutine implementations. But implementations containing these functions are rare and the `setcontext` functions also have significant shortcomings, for example with stack overflow detection.

With the advent of languages like C# and Java, coroutines went out of fashion. Recently, however, there is again a growing interest, and an increasing number of languages provide coroutines. Modern languages with coroutines include Icon, Go [12], Lua [14], Perl, Python and Ruby [10].

2.3 Implementation techniques

Coroutines have been incorporated into many different languages. Depending on the architecture of the underlying runtime system different implementation techniques have been chosen. The most common techniques for languages that use the CPU-supported stack management are:

Separate coroutine stacks A separate stack area is allocated for each coroutine. This was very simple in older operating systems (like DOS), but the introduction of exception management, the need for correctly handling stack overflows, and other stack-specific algorithms have made allocating memory that can function as stack space more difficult. Because of this, stacks also consume considerable amounts of memory, which prevents applications from using large numbers of coroutines.

Copying the stack The stack data of every coroutine is copied to and from the stack as needed. While this approach is simple to implement, it also has two main disadvantages: Firstly, the coroutine transfer becomes much more expensive (and its costs depend on the current number of stack frames). Secondly, in garbage-collected environments the data that has been copied from the stack can contain root pointers that need to be visited during garbage collection, which means that the garbage collector needs to be aware of them.

Continuations Every system that provides continuations can easily provide coroutines, by simply capturing and resuming continuations. The following example (syntax taken from [26]) shows how an implementation of symmetric coroutines could look like:

```java
public class Coroutine {
  private Continuation continuation = new Continuation();

  // returns the next scheduled coroutine
  private Coroutine next() {
    return ...;
  }

  public void yield() {
    if (continuation.capture() == Continuation.CAPTURED) {
      next().continuation.resume(null);
    }
  }
}
```

In programming languages that represent stack frames as heap objects and a stack as a linked list of stack frames, it is trivial to implement coroutines by storing the pointer to the topmost frame of the coroutine. Such systems, however, need special garbage collection algorithms to free obsolete frames.

Within the constraints of the Java programming language and a JVM it is not possible to use any of these implementation techniques, because it is not possible to gain arbitrary access to the structure of the stack. JVMs also do not provide continuation support, apart from research projects like [9] and [26]. There are, however, other techniques that can be used:

Threads as coroutines If multiple threads are synchronized in a way that allows only one of them to run at a time, and that provides explicit switching between the threads, they will appear as if they were coroutines running within one thread[2]. The drawback of this approach is that threads consume a lot of memory and that the synchronized switching between threads is an expensive operation.

Compile-time transformations A special compilation step that turns methods into state machines can be used to make methods appear like coroutines at the source level. C#'s implementation of iterator methods performs this transformation for individual methods (stackless), while frameworks like javaflow [3] perform this transformation in a way that simulates stackful coroutines. This approach leads to a significantly larger compiled code size,

[2]Only with limitations. Thread local variables, for example, will not work as expected.

and the more complicated methods leave the just in time compiler with less opportunities for optimization.

An example of the transformation of a simple C# iterator method is shown in Figure 2.3.

```
using System;
using System.Collections;

class Test {

  // The returned iterator will
  // generate 0 as the first
  // element, 1 as the second,
  // ..., and 9 as the last
  // element.
  // Then it will signal that
  // is has no more elements
  // to offer.
  static IEnumerator GetCounter() {
    for (int c = 0; c < 10; c++) {
      yield return c;
    }
  }
}
```

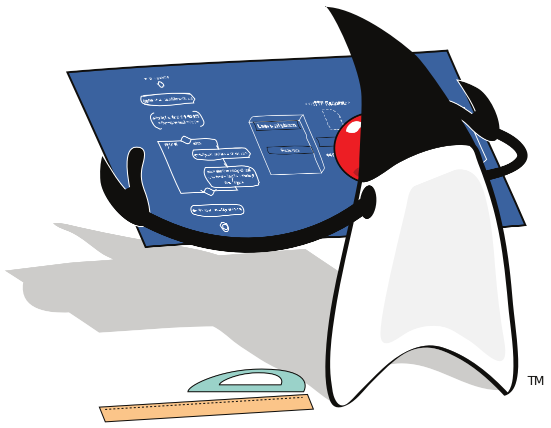
```
private class GetCounter {
  private int state = −1;
  private int count = 0;
  public int current;

  public bool MoveNext() {
    switch (state) {
      case 0:
        while (count < 10) {
          current = count;
          state = 1;
          return true;
        resume_label1:
          state = −1;
          count++;
        }
        break;
      case 1:
        goto resume_label1;
    }
    return false;
  }
}
```

Figure 2.3: A simple C# iterator method (left) and an equivalent state machine (right). Note that the simplified state machine shown here is only a pseudocode approximation. The actual result of the transformation has a length of two pages and is not representable by valid C# code.

Chapter 3

Coroutines in Java



This chapter explains the basic concepts a programmer needs to know in order to use the coroutine implementation presented in this document.

The introduction of a new programming language concept raises a number of questions that need to be answered before the new feature can be implemented:

Should new language constructs be created?

Coroutines could be implemented more elegantly by introducing new keywords like **`yield`** into the Java language. The implementation presented in this document does not do so, because this would lead to problems with backward-compatibility and would require changes to the java source compiler (`javac`), thus making the implementation more complex. Not introducing a new keyword follows the general effort of the Java language to avoid special-purpose syntax where possible.

How does the programmer access the new features' primitives?
Symmetric and asymmetric coroutines are the basic primitives of a coroutine implementation. The implementation shown here presents coroutines as instances of the `Coroutine` and `AsymCoroutine` classes. These classes contain all methods needed to perform the basic operations upon coroutines.

3.1 Thread Affinity

Operations on coroutines implicitly always operate on the current thread. In fact, it is not possible to perform any operations on the coroutines of another thread, other than thread-to-thread migration and read-only operations like querying a coroutine's status.

This allows the system to assume that only a minimal amount of synchronization is necessary, which is important in order to achieve high performance.

3.2 Stacks allocated directly from the Operating System

Most garbage collection algorithms used in JVMs perform a compaction step that moves objects in memory. Stack memory cannot be moved to another address because of locking and native stack frames, so it is not possible to allocate it from the ordinary Java heap.

Additionally, stack memory needs to be aligned on page boundaries and guarded by protected memory pages, in order to detect stack overflows. These guarantees and properties can only be acquired directly from the operating system.

The fact that the memory needs to be allocated outside of the normal Java heap makes controlling the resources of an application harder, but on the other hand this is no different than the allocation of threads - most of the memory used by a thread also lies outside of the Java memory subsystem.

3.3 Scheduling of Symmetric Coroutines

For symmetric coroutines there needs to be a scheduling strategy. In the system presented in this document symmetric coroutines are scheduled on a first come, first served basis. Each time a coroutine is executed it is moved to the end of the queue of coroutines that will be executed, and it is thus only executed again after all other coroutines have been scheduled.

While the specification of coroutines should be careful about requiring a particular behavior, this scheduling strategy seems to be a reasonable choice that can be implemented very efficiently using a doubly-linked ring of coroutines. Advancing to the next coroutines is achieved by simply moving a pointer to the next coroutine. The current coroutine will then automatically be at the end of the queue.

3.4 Input and Output Values of Asymmetric Coroutines

Asymmetric coroutines do not need to be scheduled, because they will only execute when they are called explicitly. However, it is very common for asymmetric coroutines to receive an input value to work with and/or return a result value to the caller each time they are called.

In fact, this behavior is common enough to warrant an explicit implementation, although it might also be implemented by the user on top of a simpler coroutine system. In order to make asymmetric coroutines more convenient and type-safe we specify the types of the input and output values as generic type parameters of the asymmetric coroutine class.

3.5 Thread-to-Thread Migration

Due to the thread affinity of coroutines, it is not possible to resume a coroutine on some other thread. In fact, trying to do so will lead to an exception. This is the expected behavior in the normal case and leads to good performance characteristics, but in some cases it is necessary to transfer a coroutine to some other thread.

In an application that implements user interactions as coroutines, with many user interactions running in one thread, a coroutine might be blocked due to a computationally expensive operation at one point in the user interaction. It is not acceptable for this coroutine to block all other coroutines that happen to be on the same thread, thus there needs to be a way to move coroutines to other threads.

3.5.1 Coroutine Stealing

When moving coroutines from one thread to the other there are four possible semantics depending on who takes the active role in the process:

Source and Target Thread Both the source and the target thread need to reach a synchronization point where the coroutine is handed over from one thread to the other.

Source Thread The source thread "injects" the coroutine into the target thread.

Target Thread The target thread "steals" the coroutine from the source thread.

Third Party Thread A third party thread takes the coroutine from the source thread and gives it to the target thread.

In the case of a blocked coroutine it is impractical to require the source thread to take part in the process, because it is busy and will likely not reach a synchronization point quickly. The target thread, however, is idling anyway (otherwise it would not try to acquire new coroutines), so it makes sense for it to take the active role in the process. This is called *coroutine stealing*, analogous to *work stealing*, which is an established concept in concurrent programming.

3.5.2 Migration Process

Coroutines that are moved from one thread to the other need of course be aware of the changes they will see after migration: `Thread.current()` will return the new thread, and the value of thread locals might also have changed.

Also, a coroutine cannot be transferred to another thread if there are native method frames on the stack. The VM does not have sufficient information to relocate native frames to another stack, which is required during the transfer process.

Stack frames of methods which hold locks (via the **`synchronized`** keyword) also cannot be transferred to another thread, because locks are bound to threads. Additionally, the implementation would be much more complex if locks needed to be transferred.

3.6 Serialization/Deserialization

When a coroutine has been suspended for a long time, but may be needed in the future, it might be best to move the coroutine into permanent storage (disk, DB, . . .) in order to free the memory associated with it. This is especially important in systems that sustain a large number of concurrent sessions, e.g. web servers that support many concurrent user interactions.

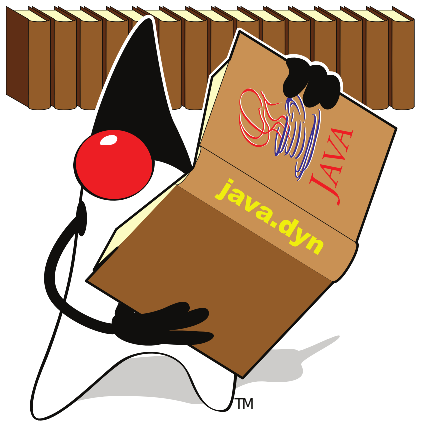
Java serialization allows a program to serialize all of its data, but it is not possible to serialize a thread. The coroutine system, on the other hand, provides a way to serialize a coroutine, which essentially turns a coroutine into an array of Java objects. Each of these objects represents one Java activation frame, and contains the frame's current method, bytecode index, local variables and expression stack. The application that uses the coroutine serialization mechanism can then decide how it wants to handle the actual serialization of these values. In a sense, the coroutine serialization mechanism does not actually perform the serialization, it only makes the contents of the coroutine's stack frames accessible to serialization.

This also means that for coroutine serialization the user has to deal with the same problems as for other uses of the Java serialization mechanism: Where to make the cut between the serialized and non-serialized parts of the program state and what to do with non-serializable objects. This is application-dependent and cannot be decided by the coroutine implementation. An application can use the object-replacement functionality provided by the `ObjectInputStream` and `ObjectOutputStream` classes to communicate its requirements to the Java serialization mechanism.

The restrictions for coroutine serialization are similar to the restrictions for thread-to-thread migration: The coroutine cannot contain native frames or frames with locked methods.

Chapter 4

Coroutine API



This chapter describes the API that is used to implement, create, use and destroy symmetric and asymmetric coroutines. This chapter also shows the API for migrating coroutines between threads and closes with a description of the API used to serialize and deserialize coroutines.

A survey of 12 different programming languages with coroutine implementations showed that there is no common naming scheme for coroutines, neither in the way the corresponding language features are called, nor in the names of the methods that are used to control them. The Java API presented here tries to be consistent with the majority of programming languages and to fit smoothly into the Java world. The API always works on the coroutines of the current thread in order to avoid having to lock data structures.

There are two main classes: `Coroutine` for symmetric coroutines, and `AsymCoroutine` for asymmetric coroutines. The size of a new coroutine's stack can be specified via the `stackSize` parameter at construction. Like the `stackSize` parameter of the `Thread` constructors this is only a hint which may or may not be obeyed by the runtime system. The system also enforces a platform-specific minimum value for this parameter.

By specifying a stack size the user can deal with extreme cases such as coroutines that call deeply recursive methods and therefore need a large stack, or large numbers of coroutines that hardly call any other methods and for which a large stack would be a waste of memory. In most cases, however, the automatically chosen stack size will be adequate.

The user can also provide the runtime with a hint that two specific coroutines should share a stack. To do so the constructor of the second coroutine needs to be called with the `sharedStack` parameter set to the first coroutine. Again, this is only a hint that may or may not be obeyed by the runtime system.

Both `Coroutine` and `AsymCoroutine` extend the class `CoroutineBase`, which contains implementation-specific code that is common to both symmetric and asymmetric coroutines.

4.1 Symmetric Coroutines

The interface for symmetric coroutines is shown in Figure 4.1. It is similar to `Thread` in that it can either be subclassed (overriding the `run()` method) or supplied with a `Runnable` at construction[1].

Each thread maintains the coroutines that are associated with it in a data structure that provides fair scheduling. The details of the scheduling algorithm are explained in Section 3.3. The coroutine system inserts each newly created `Coroutine` object after the current one, which means that it is the next coroutine to be scheduled. This also means that there is no need to register a coroutine with the runtime system, and that coroutines can only be added to the current thread. While this might seem to be an arbitrary restriction it is vital for the performance of the system, as explained in Section 3.1.

There are two ways to switch to another coroutine:

[1]`Coroutine`, however, does not implement `Runnable`. In hindsight it was a poor decision for `Thread` to do so.

```java
package java.dyn;

public class Coroutine extends CoroutineBase {

  public Coroutine();
  public Coroutine(Runnable r);
  public Coroutine(long stackSize);
  public Coroutine(Runnable r, long stackSize);
  public Coroutine(Runnable r, CoroutineBase sharedStack);

  public static void yield();
  public static void yieldTo(Coroutine target);

  public boolean isFinished();
  protected void run();
}
```

Figure 4.1: Coroutine Java API: `Coroutine` class

- **`yield`**`()` transfers control to the next coroutine, as determined by the scheduling algorithm. Under normal circumstances this means that the current coroutine will only be activated again after all other coroutines.

- **`yieldTo`**`(Coroutine target)` transfers control to the specified coroutine. The target coroutine must be a coroutine created within the current thread, otherwise an exception is thrown. The user of this method cannot make assumption as to how it influences scheduling, other than that the future calls to **`yield`**`()` will again be scheduled in a fair way.

Both these methods are static because they will often be used in a context where the current coroutine object is not readily available. They also always act on the current thread's list of coroutines, following the general paradigm that coroutines are strongly coupled to the thread they are running on. Note that the main program, i.e., the thread's initial execution context, is a coroutine as well.

The instance method `isFinished()` returns true if the `Coroutine` in question has ended by either reaching its end, by executing a **`return`**-statement or by throwing or propagating an exception.

Figure 4.2 shows an example of a simple coroutine that produces the following output:

```
start
Coroutine running 1
```

```java
public class ExampleCoroutine extends Coroutine {

  public void run() {
    System.out.println("Coroutine␣running␣1");
    yield();
    System.out.println("Coroutine␣running␣2");
  }

  public static void main(String[] args) {
    new ExampleCoroutine();
    System.out.println("start");
    yield();
    System.out.println("middle");
    yield();
    System.out.println("end");
  }
}
```

Figure 4.2: Example for the usage of `Coroutine`

```
middle
Coroutine running 2
end
```

In practice it would be better to specify the program using the `Runnable` interface, which leads to the same behavior. This is shown in Figure 4.3.

4.2 Asymmetric Coroutines

For reasons of generality, we also provide an implementation of asymmetric coroutines, called `AsymCoroutine`, shown in Figure 4.4. Instances of this class can be created by either subclassing `AsymCoroutine` (overriding the `run` method) or providing an instance of `AsymRunnable` (shown in Figure 4.5).

`AsymCoroutine` objects are not part of the ordinary `Coroutine`-scheduling, they are thus only executed when they are called explicitly. They also know their caller, which allows them to return to their caller with a `ret` call.

`AsymCoroutines` are prepared to take an input from their caller and to return an output. The types of these input and output parameters can be specified by generic type parameters. If the input and/or output parameters are not used,

```java
public class ExampleCoroutine implements Runnable {

  public void run() {
    System.out.println("Coroutine␣running␣1");
    Coroutine.yield();
    System.out.println("Coroutine␣running␣2");
  }

  public static void main(String[] args) {
    new Coroutine(new ExampleCoroutine());
    System.out.println("start");
    Coroutine.yield();
    System.out.println("middle");
    Coroutine.yield();
    System.out.println("end");
  }
}
```

Figure 4.3: Example for the usage of `Coroutine` with the `Runnable` interface

the respective type parameters should be set to `Void`. The input parameter given to the first `call` will become the `input` parameter of the run method, and the output parameter returned by the `run` method will become the last `call`'s return value.

`AsymCoroutine` and `AsymRunnable` use co- and contravariant generic type parameters in order to allow for a maximum of compatibility. An `AsymCoroutine<I, O>` can be initialized with any `AsymRunnable` that can work with any input parameter that can be cast to `I` and that will return an output parameter that can be cast to `O`. The `run` method of `AsymRunnable` has an additional parameter `coroutine`, which provides the asymmetric coroutine object needed for calls to `ret`, again with a co- and contravariant type. For example: An `AsymCoroutine<Integer, Object>` can be initialized with an `AsymRunnable<Number, String>`, because the runnable takes an input that is less specific and returns an output that is more specific.

The fact that `AsymCoroutine` implements `Iterable` allows such coroutines to be used in enhanced for loops (see Figure 4.7). In this case, the input parameter is always **`null`**, as there is no way to supply the iterator with an input.

The `AsymCoroutine` interface looks as follows:

```java
package java.dyn;

public class AsymCoroutine<I, O> extends CoroutineBase
                                 implements Iterable<O> {

  public AsymCoroutine();
  public AsymCoroutine(long stackSize);
  public AsymCoroutine(
                 AsymRunnable<? super I, ? extends O> target);
  public AsymCoroutine(
                 AsymRunnable<? super I, ? extends O> target,
                 long stackSize);
  public AsymCoroutine(
                 AsymRunnable<? super I, ? extends O> target,
                 CoroutineBase sharedStack);

  public O call(I input);
  public I ret(O output);

  public boolean isFinished();
  protected O run(I input);

  public Iterator<O> iterator();
}
```

Figure 4.4: Coroutine Java API: `AsymCoroutine` class

- `O call(I input)` transfers control to the coroutine that this method is called upon. The calling coroutine can either be a `Coroutine` or an `AsymCoroutine` instance, and it is recorded as the caller of the target coroutine. `call` passes an input parameter of type `I` to the called coroutine and returns the coroutine's output parameter, which is of type `O`. A coroutine can only be called if it is not currently in use. This means that a coroutine cannot directly or indirectly call itself.

- `I ret(O output)` suspends the current coroutine and returns to the caller. The output parameter of type `O` is passed to the calling coroutine, and the next time the current coroutine is called `ret` will return the input parameter of type `I`.

- **`boolean`** `isFinished()` returns true if the `AsymCoroutine` in question has reached its end. Invoking `call` on a `AsymCoroutine` that is not alive leads to an exception.

```java
package java.dyn;

public interface AsymRunnable<I, O> {

  public O run(AsymCoroutine<? extends I, ? super O> coro,
                                                           I value);
}
```

Figure 4.5: Coroutine Java API: `AsymRunnable` interface

It is important to note that trying to `Coroutine.yield()` to another `Coroutine` generates an exception if the current coroutine is an `AsymCoroutine`.

```java
public class CoSAXParser
             extends AsymCoroutine<Void, String> {

  public String run(Void input) {
    SAXParser parser = ...
    parser.parse(new File("content.xml"),
      new DefaultHandler() {
        public void startElement(String name) {
          ret(name);
        }
      });
    return null;
  }

  public static void main(String[] args) {
    CoSAXParser parser = new CoSAXParser();

    while (!parser.isFinished()) {
      String element = parser.call(null);
      System.out.println(element);
    }
  }
}
```

Figure 4.6: SAX parser inversion

Figure 4.6 shows an example `AsymCoroutine` that inverts a SAX parser [19] to return one XML element at a time. Figure 4.7 contains an alternative version of the main method that uses an enhanced for loop.

```java
  ...
  public static void main(String[] args) {
    CoSAXParser parser = new CoSAXParser();

    for (String element: parser) {
      System.out.println(element);
    }
  }
}
```

Figure 4.7: SAX parser inversion using enhanced for loops

4.3 Thread-to-Thread Migration

The API for thread-to-thread migration of coroutines is very simple and consists of one method that is available in both the `Coroutine` and `AsymCoroutine` classes:

```java
public boolean steal();
```

This method will migrate the coroutine it is called upon to the current thread. If the transfer was successful it will return **`true`**, otherwise it will return **`false`** or throw an exception.

The operation can fail for a number of reasons:

- It will return **`false`** if the coroutine is busy. This will happen if the coroutine is currently running or the coroutine's current thread performs an operation on it, e.g., serialization. Any use of the migration mechanism needs to be prepared for this to happen, because the migration is performed without busy waiting and on a best-effort basis.

- It will throw an `IllegalArgumentException` if:

  - The coroutine already belongs to the current thread.
  - The coroutine is the initial coroutine of its thread. The initial coroutine can never be migrated to some other thread, because this coroutine is needed in order to correctly terminate the thread.
  - The coroutine contains a stack frame of a method that holds a lock.
  - The coroutine contains a native stack frame.

  - The coroutine is an asymmetric coroutine that has not yet returned to its calling coroutine.

The distinction between a **`false`** return value and an exception is intended to differentiate between failures that can happen for any use of `steal` (because of the fact that migration is performed on a best-effort basis) and failures that will not occur for correct uses of the migration mechanism.

When a symmetric coroutine is migrated it is removed from its thread's scheduling sequence and added to the current thread's sequence as the next coroutine to be executed.

Symmetric coroutines that are migrated need to be aware that `Coroutine.yield` will now schedule different coroutines, and that calls to `Coroutine.yieldTo` that were valid before will now likely be invalid.

Symmetric and asymmetric coroutines need to be aware that calls to an asymmetric coroutine (via `AsymCoroutine.call`) that were valid before will now be invalid, unless the asymmetric coroutine was migrated as well.

4.4 Serialization / Deserialization

Serialization and deserialization is also available for both symmetric and asymmetric coroutines. The main task of these two mechanisms is to transform the stack of a coroutine into a data structure that is accessible from Java, and vice-versa. The class `CoroutineFrame`, which is used to represent stack frames, is shown in Figure 4.8 and contains the following fields:

method This field contains the method that this stack frame belongs to. It is important to note that `java.lang.reflect.Method` is not serializable. An application that serializes `CoroutineFrame` objects needs to be aware of this.

bci The bytecode index of the next instruction to execute in the method.

localCount The number of local variables in this stack frame.

```java
package java.dyn;

import java.io.Serializable;
import java.lang.reflect.Method;

public class CoroutineFrame implements Serializable {

  public Method method;
  public int bci;

  public int localCount;
  public int expressionCount;

  public long[] scalarValues;
  public Object[] objectValues;

  public CoroutineFrame(Method method, int bci,
     int localCount, int expressionCount, long[] scalarValues,
                                      Object[] objectValues);
}
```

Figure 4.8: Coroutine Java API: `CoroutineFrame` class

expressionCount The number of stack slots on the expression stack that were used for evaluating an expression at the time the coroutine was suspended. A stack frame will only contain temporary expressions if the coroutine was suspended while calculating an expression, i.e., during a function call.

scalarValues This array of length `localCount + expressionCount` contains the scalar values of all local variables and the expression stack. Entries that contain an object reference have no scalar value, and the `scalarValues` array contains a zero value in that case.

objectValues This array is of the same length as `scalarValues` and contains the object references of all local variables and the expression stack. Entries that contain a scalar value have no object reference, and the `objectValues` array contains a null value in that case.

The serialization mechanism can of course not only be used for storing coroutines to disk. In fact, the ability to modify the stack of a running application can be useful for dynamic languages that compile to bytecode on the fly (Ruby, JavaScript, etc.): If the language runtime detects that a piece of code that is executed by an interpreter loop is taking too long, it can suspend the execution, replace the interpreter loop with compiled (to bytecode) methods and resume the execution. This allows languages that run on top of a JVM to perform on-stack replacement at the Java level, similar to the JVM itself, which performs on-stack replacement at the native code level.

Arbitrarily modifying stack frames of course requires an application to have extensive knowledge about the structure of the compiled methods, which is not provided by the coroutine implementation.

An example that uses the serialization mechanism to perform session persistence is shown in Section 5.2.

4.4.1 Serialization

The serialization of coroutines is supported via the following method that is available in both symmetric and asymmetric coroutines:

```
public CoroutineFrame[] serialize();
```

If the coroutine contains stack frames of native methods or stack frames that currently hold locks, which cannot be transformed into `CoroutineFrame` objects, it will throw an `IllegalArgumentException`. It will also throw an exception if the coroutine is the initial coroutine of a thread, which cannot be serialized because it will always contain native frames.

If successful, this method will return an array of `CoroutineFrame` objects. If the coroutine's stack contained references to Java objects, then the resulting `CoroutineFrame` objects will contain references to the same objects, as shown in Figure 4.9.

In the simplest case an application can serialize the array of `CoroutineFrame` objects using an `ObjectOutputStream`. It only needs to take care of non-serializable classes, as shown in Section 5.2.

In a more complex application it might be necessary to perform additional substitutions, e.g., replace objects that represent database entities with a serializable form or take care of other objects that are not serializable. Applications that keep coroutines in a serialized state for only a limited amount of time might associate IDs with non-serializable objects, and store these objects in a hash table or a similar data structure.

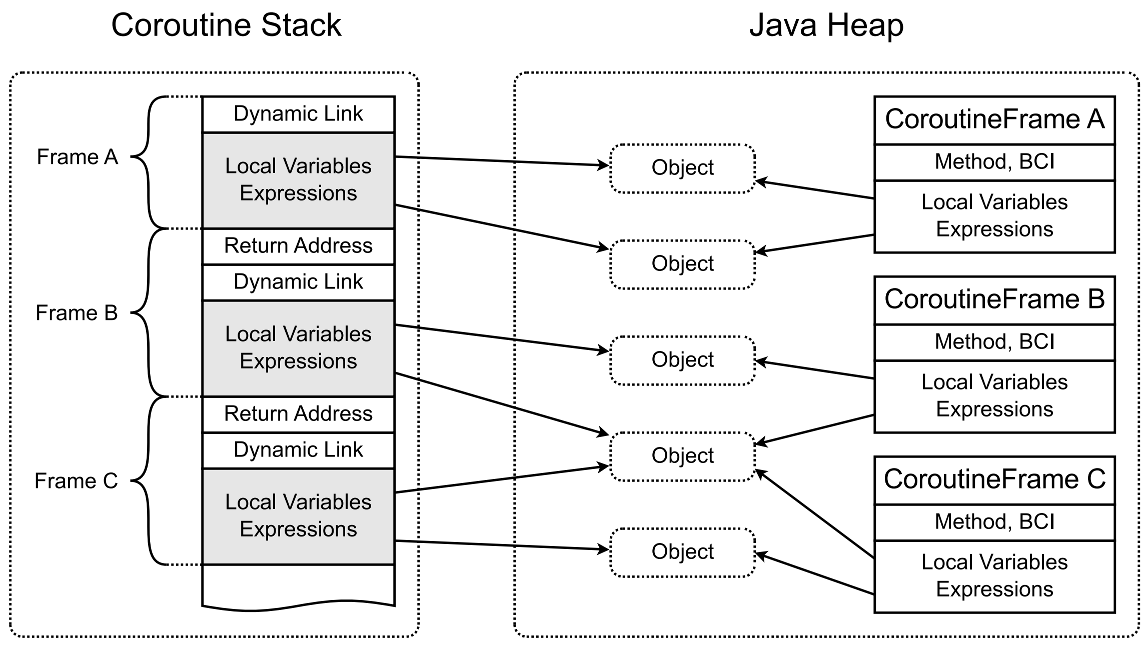


Figure 4.9: A coroutine stack (left) and the equivalent `CoroutineFrame` objects (right).

4.4.2 Deserialization

Similar to serialization the deserialization is also performed by a method that is available in both symmetric and asymmetric coroutines:

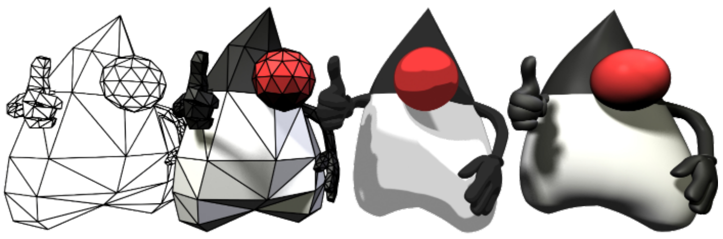
```
public void deserialize(CoroutineFrame[] frames);
```

This method is the opposite of `serialize`: it takes an array of `CoroutineFrame` objects and replaces the target coroutine's stack with the given frames. It will fail if the coroutine is the thread's initial coroutine, in which case it will throw an `IllegalArgumentException`.

Again, if the `CoroutineFrame` objects contain references to Java objects, then the coroutine's stack frames will also contain references to these objects, as shown in Figure 4.9.

Chapter 5

Case Studies and Examples



This chapter presents a number of examples that both motivate and explain the usage of coroutines for real world applications. Although the examples are only small in size, they are sufficient to show the advantages of coroutines for numerous tasks. They also motivate the more advanced concepts of thread-to-thread migration and serialization.

5.1 Machine Control

Machine control programs typically have a fixed cycle time, for example 1 ms. In each cycle all control programs for the different components are executed exactly once. Some of them are simple and perform almost the same actions in every cycle, while others are more complex and need to perform a series of actions over a longer period of time. In general every component only executes a few instructions before switching to the next one, so that the total execution time is influenced heavily by the time it takes to perform a switch.

The following example is influenced by the author's experience with the Monaco programming language [21], which is a domain-specific language for machine control applications.

The example program controls an injection molding machine, which is used to produce parts from thermoplastic materials by injecting molten plastic into a mold cavity, letting it cool and ejecting the finished part.

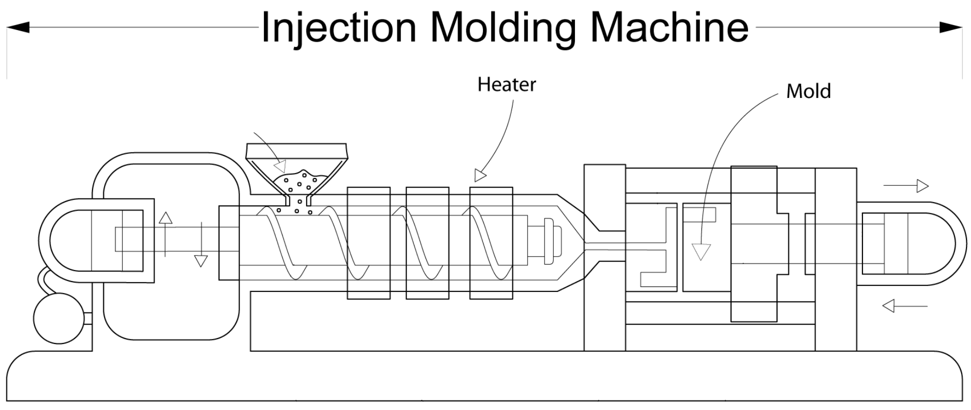


Figure 5.1: Schematic of an injection molding machine
Created by Brendan Rockey, University of Alberta Industrial Design. License: CC BY 3.0

The control program in this simplified example needs to control the following components of the molding machine (see Figure 5.1):

Heater The heater controls the temperature of the molten material. In order to produce faultless parts the temperature needs to be within a specified range.

Mold and Injection Unit The injection unit starts and stops the injection of material. It also controls the position of the mold and the ejection actuator.

5.1.1 Implementation

The components are accessed via the following two interfaces `Heater` and `Injection`:
(the actual implementation, which is machine-specific, is not presented here)

Heater.java - Interface for the heater component

```java
public interface Heater {

  public double getTemperature();

  public void enable();
  public void disable();
}
```

This interface can be used to determine the current temperature of the heater, and to enable and disable the heating component.

HeaterControl.java - Control program for the heater component

```java
public class HeaterControl extends Coroutine {
  private final Heater heater;

  public HeaterControl(Heater heater) {
    this.heater = heater;
  }

  protected void run() {
    while (true) {
      // wait until temperature falls below lower threshold
      while (heater.getTemperature() >= 75)
        Coroutine.yield();

      // enable heater
      heater.enable();

      // wait until temperature is at upper threshold
      while (heater.getTemperature() < 80)
        Coroutine.yield();

      // disable heater
      heater.disable();
    }
  }
}
```

The coroutine `HeaterControl` is responsible for keeping the heater at the correct temperature. It is implemented as a simple loop that waits for the temperature to drop below 75°C. The heater is then enabled until the temperature reaches 80°C.

Injection.java - Interface for the injection component

```java
public interface Injection {

  public double getVolume();

  public double getPosition();

  public void setTargetPosition(double pos);

  public void startInjection();
  public void stopInjection();

  public void ejectPart();
}
```

The `Injection` interface is used to determine the total amount (volume) of material that was injected, to determine the position of the movable part of the mold (0 = closed, 100 = open), to set the target position for the mold (the mold will immediately start to move towards the given position), to start and stop the injection of material and to eject the finished part from the machine.

InjectionControl.java - Control program for the injection component

```java
public class InjectionControl extends Coroutine {
  private final Injection injection;
  private final Heater heater;

  public InjectionControl(Injection injection, Heater heat) {
    this.injection = injection;
    this.heater = heat;
  }

  protected void run() {
    double initialVolume = injection.getVolume();

    // close the mold
    injection.setTargetPosition(0);
    while (injection.getPosition() > 0)
      Coroutine.yield();

    // wait until the heater has reached the minimum
    // temperature
    while (heater.getTemperature() < 70)
      Coroutine.yield();

    // start injecting
    injection.startInjection();

    // wait until 100 units have been injected, then stop
    while (injection.getVolume() < initialVolume + 100)
```

```
      Coroutine.yield();
    injection.stopInjection();

    // wait 10 cycles, i.e., 10 ms
    for (int i = 0; i < 10; i++)
      Coroutine.yield();

    // open the mold
    injection.setTargetPosition(100);
    while(injection.getPosition() < 100)
      Coroutine.yield();

    // finally: eject the new part
    injection.ejectPart();
  }
}
```

The coroutine `InjectionControl` performs a more complex series of actions. It first closes the mold, waiting for the movement to be finished. Then it waits for the heater to reach a temperature of at least 70°C, after which it can start the injection process.

As soon as 100 units of material have been injected the injection is stopped. After a cooling time of 10 cycles, i.e., 10 ms, the mold is opened again, and when this is done the finished part is ejected.

5.1.2 Conclusion

This example shows that coroutines can be a very natural and convenient way to write code that performs a complex series of actions. The coroutines in this example also perform non-blocking busy waiting.

Using threads in this context would not be ideal: Real-world systems can have hundreds of components, each of which would need a separate thread. These threads would have to be synchronized in order to pass control from one to the other. This synchronized thread switch is a very expensive operation. Machine control programs normally run at a CPU load of 10-20%, and with hundreds of switches within one 1 ms cycle the switching time will dominate the system performance.

In contrast to threads, the existence of hundreds of coroutines is not a performance problem, because coroutines use less resources than threads and perform switches by an order of magnitude faster than threads. Also, synchronization is unnecessary with coroutines.

As with any coroutine this example could also have been implemented using explicit state machines, but this would have made it much more complex and harder to understand.

This example could also have been implemented even more elegantly if Java had an implementation of lambda expressions [15], which is currently planned for JDK 8 [16].
Instead of:

```
while (heater.getTemperature() >= 75)
  Coroutine.yield();
```

one could simply write:

```
waitUntil(#{heater.getTemperature() < 75});
```

5.2 Session Persistence

Servers that implement user sessions often have to solve the problem of how to deal with a large number of concurrent sessions. For some types of systems there will likely by a large amount of time between subsequent interactions with the user. It may be impractical to keep all sessions in memory until they expire, so there needs to be a way to store these sessions, e.g. to disk or to a database.

The following example shows a simple user interaction that is implemented using coroutines, which can be suspended to disk. Although it is a simplified example, it still demonstrates how the serialization of sessions can be achieved.

A typical interaction with the main program, called `Questions`, looks like this:

```
> java Questions
Please enter your name:

> java Questions "John Doe"
Please enter your country:

> java Questions Genovia
Please enter your age:

> java Questions twenty—eight
Please enter your age:
```

```
Please enter a valid number!

> java Questions 28
Please enter your email address:

> java Questions no@spam.at
Name: John Doe, Country: Genovia, Age: 28, Email: no@spam.at
```

While the interaction in this example takes place via command-line parameters, the solution can also be applied to other types of interaction, like web servers.

5.2.1 Implementation

The implementation is divided into three parts: The class that models the user interaction, the main program that drives the serialization and deserialization and helper classes that are needed to serialize the coroutine.

Interaction.java - Implementation of the user interaction

```java
public class Interaction
      implements AsymRunnable<String, String>, Serializable {

  private AsymCoroutine<? extends String, ? super String> coro;

  //helper method used to acquire user input
  private String readString(String message) {
    return coro.ret(message);
  }

  //helper method used to acquire numerical user input
  private int readNumber(String message) {
    int tries = 0;
    while (true) {
      try {
        return Integer.valueOf(readString(message));
      } catch (NumberFormatException e) {
        if (tries++ == 0)
          message += "\nPlease␣enter␣a␣valid␣number!";
      }
    }
  }

  @Override
  public String run(
          AsymCoroutine<? extends String, ? super String> coro,
          String value) {

```

```java
    this.coro = coro;

    String name = readString("Please␣enter␣your␣name:␣");
    String country = readString("Please␣enter␣your␣country:");
    int age = readNumber("Please␣enter␣your␣age:");

    if (age < 70) {
      String email;
      email = readString("Please␣enter␣your␣email␣address:");
      return "Name:␣" + name + ",␣Country:␣" + country +
             ",␣Age:␣" + age + ",␣Email:␣" + email;
    } else {
      String telephone = readString("Please␣enter␣your␣" +
                                    "telephone␣number:");
      return "Name:␣" + name + ",␣Country:␣" + country +
             ",␣Age:␣" + age + ",␣Telephone:␣" + telephone;
    }
  }
}
```

This is the only class a programmer would have to implement within a framework that supports the implementation of sessions via coroutines. It is programmed as if the user input was available immediately any time the user is presented with a question.

The user input is forwarded to the asymmetric coroutine via its input value, and the console output is passed to the main program via the coroutine's output value.

The `readString` method would, in a more sophisticated version, employ some form of non-blocking i/o. The coroutine would then be restored whenever user input is available.

It is important to note that local variables are available again when the coroutine is restored and that coroutines can be suspended from within subsequently called methods (like `readNumber` and `readString`).

Questions.java - Main program containing the serialization / deserialization logic

```java
public class Questions {

  public static void main(String[] args) {

    AsymCoroutine<String, String> coro;

    // check if there is a suspended coroutine on disk
    if (new File("data.bin").exists()) {
      FileInputStream fis = new FileInputStream("data.bin");
      CoroutineInputStream<String, String> cis;
      cis = new CoroutineInputStream<String, String>(fis);
      coro = cis.readAsymCoroutine();
      cis.close();
    } else {
      coro = new AsymCoroutine<String, String>(
                                                new Interaction());
    }

    // transfer control, this will execute the next step
    String msg = coro.call(args.length > 0 ? args[0] : null);
    System.out.println(msg);

    // if the coroutine isn't finished: store to disk
    if (coro.isFinished()) {
      new File("data.bin").delete();
    } else {
      FileOutputStream fos = new FileOutputStream("data.bin");
      CoroutineOutputStream<String, String> cos;
      cos = new CoroutineOutputStream<String, String>(fos);
      cos.writeAsymCoroutine(coro);
      cos.close();
    }
  }
}
```

This main program contains the main serialization / deserialization logic.

It checks whether there is a suspended coroutine on disk (stored in a file called `data.bin`), and if there is one it restores it using the coroutine deserialization mechanism. If there is no suspended coroutine then a new one will be created.

The main program then transfers control to the asymmetric coroutine, which will perform one step in the user interaction.

Finally, if the coroutine (and thus, the user interaction) is not finished yet it is serialized to disk. Otherwise, the data file is deleted.

CoroutineOutputStream.java - Object output stream with special coroutines handling

```java
public class CoroutineOutputStream<I, O>
                                        extends ObjectOutputStream {

  private AsymCoroutine<I, O> coro;

  public CoroutineOutputStream(OutputStream out) {
    super(out);
    enableReplaceObject(true);
  }

  @Override
  protected Object replaceObject(Object obj) {
    if (obj instanceof Method) {
      return new SerializableMethod((Method) obj);
    } else if (obj == coro) {
      return new CoroutineDummy();
    }
    return super.replaceObject(obj);
  }

  public void writeAsymCoroutine(AsymCoroutine<I, O> coro) {
    this.coro = coro;
    writeObject(coro.serialize());
  }
}
```

The special object output stream `CoroutineOutputStream` is needed for two reasons: It needs to replace instances of `java.lang.reflect.Method`, which are not serializable, with instances of the `SerializableMethod` class, and it needs to replace all instances of the coroutine in question with an instance of the `CoroutineDummy` class (because coroutine objects themselves are also not serializable).

CoroutineDummy.java - Helper class used to represent serialized coroutine objects

```
public class CoroutineDummy implements Serializable {
}
```

The helper class `CoroutineDummy` is needed because only the `CoroutineFrame` objects are serializable, and not the `Coroutine` and `AsymCoroutine` objects themselves.

SerializableMethod.java - Helper class to replace `java.lang.reflect.Method`

```
public class SerializableMethod implements Serializable {

  private final Class<?> clazz;
  private final String name;
  private final Class<?>[] parameters;

  public SerializableMethod(Method method) {
    this.clazz = method.getDeclaringClass();
    this.name = method.getName();
    this.parameters = method.getParameterTypes();
  }

```

```java
  public Method getMethod() {
    return clazz.getDeclaredMethod(name, parameters);
  }
}
```

The serializable class `SerializableMethod` is used by the coroutine object output stream to store references to instances of `java.lang.reflect.Method`. This implementation might not be enough for complex use cases, but it is a simple workaround for the fact that the reflective method class is not serializable.

CoroutineInputStream.java - Object input stream with special handling of coroutines

```java
public class CoroutineInputStream<I, O>
                                   extends ObjectInputStream {

  private AsymCoroutine<I, O> coro;

  public CoroutineInputStream(InputStream in) {
    super(in);
    enableResolveObject(true);
  }

  @Override
  protected Object resolveObject(Object obj) {
    if (obj instanceof SerializableMethod) {
      return ((SerializableMethod) obj).getMethod();
    } else if (obj instanceof CoroutineDummy) {
      return coro;
    }
    return super.resolveObject(obj);
  }

  public AsymCoroutine<I, O> readAsymCoroutine() {
    coro = new AsymCoroutine<I, O>();
    CoroutineFrame[] frames = (CoroutineFrame[]) readObject();
    coro.deserialize(frames);
    return coro;
  }
}
```

The special object input stream `CoroutineInputStream` undoes the replacements performed by the coroutine object output stream: It transforms instances of `SerializableMethod` back into reflective method objects and replaces `CoroutineDummy` instances with a new coroutine object.

5.2.2 Conclusion

Being able to serialize coroutines is a vital feature for numerous applications such as web frameworks or other systems with long-lasting user sessions. At the same time it will present the implementors of frameworks that are built on top of a coroutine system with a number of challenges: How should non-serializable classes (like `java.lang.Method`) be dealt with, and how deep should the object graph be serialized? The replacement feature of Java object serialization streams is vital in solving these problems.

Chapter 6

Related Work

Most of the publications on coroutines deal with the operational semantics and the connection to other research areas, while the actual implementation details are often omitted.

Weatherly et al. [28] implemented coroutines in Java for simulation tasks. Their implementation copies data to and from the stack for each context switch. The focus of their work lies on the connection to the simulation infrastructure, and they hardly touch the intricacies of the coroutine semantics and implementation in the Java environment.

Ierusalimschy et al. [14] describe how the Lua virtual machine handles coroutines. Their implementation uses a separate stack for each coroutine. The Lua virtual machine is interpreter-only, and the interpreter works in a *stackless* way. This means that the actual stack data can be stored at an arbitrary memory position, completely circumventing the problem of dealing with native stacks.

Bobrow and Wegbreit [4] show an implementation of coroutines that uses a "spaghetti stack" containing stack frames of multiple coroutines combined into a single stack. Every time a subroutine is called the system has to check if there is enough space for a new stack frame, because some other coroutine might occupy the area where the stack frame would normally be placed. While this approach can be efficient, it suffers from a number of drawbacks: It requires extensive modifications within the compiler and the runtime system, performs costly range checks for each method call and violates the assumption made by modern operating systems that there is always a certain amount of available stack space beneath the current stack frame.

Pauli and Soffa [20] improve upon Bobrow and Wegbreit. They introduce a dynamic and moving reentry point for coroutines, instead of the static one used by the original algorithm. They also show how to apply the algorithm to statically scoped languages and conclude with a detailed analysis of different variations of the algorithm. However, all models described by Pauli and Soffa still incur a significant overhead for each method call.

Mateu [17] makes the case that a limited version of continuations can be implemented via coroutines. He presents an implementation that heap-allocates frames and uses a generational garbage collector to free obsolete frames. The performance measurements show that it is difficult to find the optimal size of this frame heap because the performance is dominated by the trade-off between CPU cache efficiency (small heap) and fewer collections (large heap).

Carroll [5] shows an example of a real-world problem that would benefit from a coroutine implementation in Java. Two consecutive parsers are used to parse RDF (which is based on XML). Using two ordinary parsers is only possible if they are executed in different threads. Carroll shows that a significant performance increase is possible if the two parsers are combined into a single thread. In this case this is achieved via inverting the RDF parser, which is undesirable from a design perspective. The need for manual inversion could be avoided if real coroutines were available.

Second Life, developed by Linden Lab, makes heavy use of the concept of portable agents. These portable agent scripts [7], which implement the behavior of virtual objects, are essentially coroutines that can be transfered to other VMs whenever the virtual object passes the boundary of a simulation server (called *sim*). Since August 2009 Second Life executes these agent scripts on Mono, an open source implementation of the Microsoft .NET platform. In order to be able to suspend coroutines and move running scripts from server to server, the scripts are instrumented in such a way that they can store and restore their current state, similar to the javaflow library [3]. These transformation-based coroutines are at least an order of magnitude slower than a native coroutine or continuation implementation [26].

Bibliography

[1] Adams, IV, N. I., D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, M. Wand, and H. Abelson: *Revised^5 report on the algorithmic language Scheme.* ACM SIGPLAN Notices, 33(9):26–76, 1998. `ftp://ftp.cs.indiana.edu/pub/scheme-repository/doc/standards/r5rs-html.tar.gz`.

[2] Adya, Atul, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur: *Cooperative task management without manual stack management.* In *Proceedings of the USENIX Annual Technical Conference*, pages 289–302, Berkeley, CA, USA, Jun 2002. USENIX Association.

[3] Apache Commons: *Javaflow*, 2009. `http://commons.apache.org/sandbox/javaflow/`.

[4] Bobrow, D. G. and B. Wegbreit: *A model and stack implementation of multiple environments.* Communications of the ACM, 16(10):591–603, 1973.

[5] Carroll, Jeremy J.: *Coparsing of RDF & XML.* Technical Report HPL-2001-292, Hewlett Packard Laboratories, Nov 2001. `http://www.hpl.hp.com/techreports/2001/HPL-2001-292.pdf`.

[6] Conway, Melvin E.: *Design of a separable transition-diagram compiler.* Communications of the ACM, 6(7):396–408, 1963.

[7] Cox, Robert and Patricia Crowther: *A review of linden scripting language and its role in second life.* In Purvis, Maryam and Bastin Savarimuthu (editors): *Computer-Mediated Social Networking*, volume 5322 of *Lecture Notes in Computer Science*, pages 35–47. Springer Berlin / Heidelberg, 2009.

[8] Dahl, Ole Johan, Bjorn Myrhaug, and Kristen Nygaard: *SIMULA 67. common base language.* Technical report Publ. No. S-2, Norwegian Computing Center, Oslo, Norway, May 1968. Revised Edition: Publication No. S-22.

[9] Dragoş, Iulian, Antonio Cunei, and Jan Vitek: *Continuations in the Java virtual machine.* In *International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems.* Technische Universität Berlin, 2007.

[10] Flanagan, David and Yukihiro Matsumoto: *The Ruby programming language.* O'Reilly, 2008, ISBN 978-0-59651-617-8.

[11] Goldberg, Adele and David Robson: *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[12] Google: *Go programming language*, 2009. `http://golang.org/`.

[13] Griswold, Ralph E. and Madge T. Griswold: *The Icon programming language.* Peer-to-Peer Communications, San Jose, CA, USA, third edition, 1997, ISBN 1-57398-001-3. `http://www.cs.arizona.edu/icon/lb3.htm`.

[14] Ierusalimschy, Roberto, Luiz Henrique de Figueiredo, and Waldemar Celes: *The implementation of Lua 5.0.* Journal of Universal Computer Science, 11(7):1159–1176, 2005.

[15] Java Community Process: *JSR 335: Lambda Expressions for the Java^{TM}Programming Language*, 2010. `http://jcp.org/en/jsr/detail?id=335`.

[16] Java Community Process: *JSR 337: Java^{TM}SE 8 Release Contents*, 2010. `http://jcp.org/en/jsr/detail?id=337`.

[17] Mateu, Luis: *An efficient implementation for coroutines.* In *Proceedings of the International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, pages 230–247. Springer, 1992.

[18] Moura, Ana Lúcia De and Roberto Ierusalimschy: *Revisiting coroutines.* ACM Transactions on Programming Languages and Systems, 31(2):6:1–6:31, February 2009.

[19] Murray-Rust, Peter, David Megginson, Tim Bray, *et al.*: *Simple API for XML*, 2004. `http://www.saxproject.org/`.

[20] Pauli, W. and Mary Lou Soffa: *Coroutine behaviour and implementation.* Software—Practice and Experience, 10(3):189–204, March 1980.

[21] Prähofer, Herbert, Dominik Hurnaus, Roland Schatz, and Hanspeter Mössenböck: *The domain-specific language monaco and its visual interactive programming environment.* In *Visual Languages and Human-Centric Computing, 2007. VL/HCC 2007. IEEE Symposium on*, pages 104–110, 2007.

[22] Python Software Foundation: *Greenlet - Lightweight in-process concurrent programming*, 2010. `http://pypi.python.org/pypi/greenlet`.

[23] Shankar, Ajai: *Implementing Coroutines for .NET by Wrapping the Unmanaged Fiber API*, 2005. `http://msdn.microsoft.com/en-us/magazine/cc164086.aspx`.

[24] Simon, Doug, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek White: *Java on the bare metal of wireless sensor devices: the squawk java virtual machine.* In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pages 78–88, New York, NY, USA, 2006. ACM, ISBN 1-59593-332-8. `http://doi.acm.org/10.1145/1134760.1134773`.

[25] Stadler, Lukas: *Serializable Coroutines for the HotSpot^{TM} Java Virtual Machine.* Master's thesis, Johannes Kepler University Linz, Austria, 2011.

[26] Stadler, Lukas, Christian Wimmer, Thomas Würthinger, Hanspeter Mössenböck, and John Rose: *Lazy Continuations for Java Virtual Machines.* In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 143–152. ACM, 2009. `http://doi.acm.org/10.1145/1596655.1596679`.

[27] TIOBE Software BV: *Tiobe programming community index*, April 2010. `http://www.tiobe.com/tpci.htm`.

[28] Weatherly, R. M. and E. H. Page: *Efficient process interaction simulation in Java: Implementing co-routines within a single Java thread.* Winter Simulation Conference, 2:1437–1443, 2004.

[29] Wirth, Nikolaus: *Programming in Modula-2.* Texts and Monographs in Computer Science. Springer-Verlag, Berlin, 4th edition, 1988, ISBN 0-387-50150-9.