# Continuations in the Java Virtual Machine

Iulian Dragoş[1], Antonio Cunei[2], and Jan Vitek[2]

[1] École Polytechnique Fédérale de Lausanne, Switzerland
[2] Purdue University, USA

**Abstract.** Continuations have received considerable attention lately as a possible solution to web application development. Other uses proposed in the past, such as cooperative threading, coroutines or writing iterators, made them an attractive feature of dynamically typed languages. We present issues involved in adding continuations to a statically typed, object-oriented language like Java, and its implementation in the Java Virtual Machine. We propose three different flavors of continuations, and study their interactions with the base language, focusing on Java's concurrency model. We describe our implementation in Ovm, a realtime Java Virtual Machine, and discuss open issues.

## 1 Overview

Continuations are a way to represent the "rest of the computation" at a given point in the program [1]. Most languages that have first-class continuations represent them as functions that, when called, cause the immediate transfer of control from the caller to the point where it was *captured*. The computation will resume in the same state as when it was captured. It has to be noted that, unlike *setjmp/longjmp* in C, a continuation can be called at any point during program execution, sometimes long after the activation frame where it was captured has been left.

Continuations are used to encode coroutines, cooperative threading, to write iterators (C# iterators or Python generators) and have been proposed as a natural abstraction for interactive web applications [2, 3]. For instance, a tree traversal iterator, which might be a non-trivial task to code in plain Java, can be as simple as shown in Fig. 1. `Yield` suspends the current method and returns a value to the caller. Subsequent calls to `next` resume the traversal immediately after the previous call to `yield`. This method could not be coded in Java without continuations. For brevity, we don't include the definition of this method.

Our goal when designing continuations for Java was to confine changes to the virtual machine. Previous work has shown that continuations can be added in languages that target uncooperative virtual machines like the JVM or the .NET [4, 5]. However, implementing continuations in the virtual machine is likely to be more efficient and make them available to all languages targeting that platform. Furthermore, we chose to expose them through library functions rather than new opcodes, so that existing compilers need not be modified to take advantage of this new feature.

```
public class System {
  public static Object callcc(
    Runnable1 r);
  public static Object callccBounded(
    ContinuationBound cb, Runnable1 r);
  public static Object callccOneShot(
    Runnable1 r);
  //..
}

public interface Runnable1 {
  public void run(Continuation k);
}

public final class Continuation {
  public void call(Object v);
}
```

(a) Additions to the java.lang package.

```
public Iterator getIterator() {
  return new Generator() {
    protected void generate() {
      preorder(Tree.this);
    }

    private void preorder(Tree t) {
      if (t.isLeaf())
        yield(t.value);
      else {
        preorder(t.left);
        preorder(t.right);
      }
    }};
}
```

(b) A tree iterator.

**Fig. 1.**

Continuations are exposed through the library function `callcc`[3]. This follows common practice in languages that have first-class continuations (Scheme, Smalltalk, Ruby, ML, etc.). Fig. 2 shows an example of using `callcc` in our system. The list of additions to the `java.lang` package is shown in Fig. 1 (a different class than `System` could have been used for adding `callcc` but we chose to make it clear it is a VM service).

```
System.callcc(new Runnable1() {
  public void run(Continuation k) {
    k.call(new Integer(42));
  }
});
```

**Fig. 2.** Using callcc

### 1.1 Call/cc and Call

We follow standard semantics for the call/cc mechanism. In the following text the term "execution context" will refer to the call stack, local variables and instruction pointer during the execution of a method. Global state and objects on the heap are not saved/restored by continuations.

[3] Call with current continuation

**System.callcc** This method captures a continuation `k` which represents the current execution context, with the instruction pointer pointing immediately after this call. It then proceeds to call method `Runnable1.run` on its parameter.
Return value: If the call to `Runnable1.run` finishes without a call to `k.call` being made, `System.callcc` returns `null`. Otherwise, it returns the value passed as argument to `k.call` (see below).

**Continuation.call** A call to this method restores the execution context captured by the given continuation. The current execution context is dropped and execution continues at the point where `System.callcc` was called, returning the value passed as argument to `call`. This method never returns.

## 2 Continuations

There are several flavours of continuations presented in literature [6–8]. We have chosen to implement three kinds of continuations: full continuations for their expressivity, one–shot continuations because they are fast and serve a common use–case, and delimited continuations as a good compromise between efficiency and expressivity.

**First–class continuations** This is the most general kind of continuations. There are no limitations with regard to the number of times a continuation can be called, or its lifetime. Since they need to copy the full execution stack, they are more costly than the other kinds of continuations.

**Delimited continuations** The user first obtains a *continuation bound* which is then used to obtain a delimited continuation. Such a continuation is valid for as long as the execution stack does not exit its bound. An invalid continuation throws an exception when called. Valid delimited continuations can still be called any number of times. A delimited continuation costs less than a full continuation, since they only need to copy a subsection of the execution stack, given by the continuation bound and the point where `callcc` is called.

**One–shot continuations** A one shot continuation can be called a single time. They are valid for as long as the execution does not leave the `callcc` frame[4]. They are the "cheapest" continuations available, since the system only has to save the instruction pointer.

The user has the choice of which kind of continuations to use by calling the corresponding `callcc` method. Full continuations can be used when the whole execution context needs to be saved and restored, for instance to code threads. One–shot continuations can be used fast for non–local returns while delimited continuations are a good choice when only a limited part of the execution context should be saved/restored (for instance, to write coroutines).

[4] This can be regarded as an implicit application of that continuation by callcc itself, after the runnable that was passed as argument returns.

## 3 Interactions with the Base Language

Existing Java language features turn out to interact in unexpected ways with continuations. Exceptions, language support for synchronization through monitors, threads and the Java security model all present possible issues when continuations are added to the language. We will consider each one in turn and present the problems we identified and our solutions.

### 3.1 Exceptions

The try-catch-finally statement raises issues in the presence of continuations: *finally* handlers are guaranteed to run, be it through normal or abrupt (exception thrown) completion of its protected block. Consider the code shown in Fig. 3 (a).

```
Handle h = getNativeHandle();
try {
 // use h in various ways
 System.callcc(
   new Runnable1() {
     public void run(Continuation k) {
       this.dangerousK = k;
     }});
 // use h again
} finally {
  releaseNativeHandle(h);
}
```

(a)

```
System.callcc(new Runnable1() {
  public void run(Continuation k) {
    this.k1 = k;
    try {
      System.callcc(new Runnable1() {
        public void run(Continuation k) {
          this.k2 = k;
          //..
          k1.call();
        }});
      // k2 points here
    } finally {
      //..
    }
    }});
// k1 points here
```

(b)

**Fig. 3.** Continuations and "finally"

In usual Java, this code ensures the native handler is released under all circumstances. It also saves a continuation which can later be called, and make use of the handle *after* it has been released. Furthermore, it will reach the end of the try–finally block and run the handler once more — releasing the native handler a second time.

The problem comes from the fact that `callcc` breaks the assumption that a *finally* handler is run just once. The converse, calling a continuation while *finally* handlers are active has a similar problem: should they be run? The answer is not easy, since it depends on the continuation: in Fig. 3 (b) calling `k1` leaves the *finally* handler, while calling `k2` does not.

Additionally, a VM-only solution cannot be adopted, since the Java VM [9] has no notion of *finally* handlers. It is the compiler's job to generate proper code

on all control flow paths to invoke the finally code. Thus, Java with continuations has to either forbid such cases, or relax the guarantees of `finally`. To forbid them, a simple extension to the type checker could ensure that all methods that call `callcc` or `Continuation.call` are annotated with a special annotation, and that no such methods are called from within blocks protected by a `finally` handler. Relaxing the guarantees for `finally` is the road taken by Ruby and Smalltalk, who will not honor their equivalent of `finally` when continuations are involved.

An interesting alternative to try-finally is Scheme's `dynamic-wind` [10], which works like a try-finally with a prelude: whenever control enters the block (either normally or through a continuation), the *prelude* is run; the same goes for leaving the block and the *postlude*. Such a method can be easily written when having `callcc` [11].

The best way to deal with continuations in the presence of Java's try-finally is an interesting research question, and has to be dealt with by any implementation that adds continuation to the language. However, our focus in this paper is the VM side of things, so we will not explore further.

### 3.2 Synchronization

Capturing a continuation inside a `synchronized` block means that such a block can be re-entered any number of times through that continuation. Code inside that block assumes that a number of locks have been acquired and therefore, whenever it executes, they have to actually be held.

Our solution is that `Continuation.call` fails (with an `IllegalMonitorState exception`) if the current thread does not hold exactly the same monitors as the one which captured the continuation. To see why, we need to notice that finalizers and monitors are closely related. A `synchronized` block implicitly defines a finally handler which will release that monitor. It follows that we have the following restrictions on callers of a continuation captured inside a synchronized block:

**Destination compatibility** The caller must be able to release at least the same monitors as the ones active at the point of capture. This is because it will run the special finally handlers.

**Source compatibility** The caller must not hold monitors other than the ones active at the point of capture. This is because it will drop the current execution context, and its own finally handlers will not be executed leading to unreleased monitors.

It follows that the two sets of monitors have to be equal. Our implementation keeps track of the monitors acquired by a thread and makes the necessary checks when a continuation si captured or applied. User code can test whether a continuation is valid in the current context by calling `Continuation.isValid`. This problem seems to have gone unnoticed by the creators of RiFE [5].

### 3.3 Threads

Continuations refer to the thread that created them. What happens if a thread calls a continuation captured by another thread? Delimited and one–shot continuations fail at runtime. Since they carry only a part of the execution context, they can't recreate it on the target thread. Full continuations could, in principle, be called in a different thread than the one who captured them (our current implementation does not allow this).

### 3.4 Java Security Model

The Java security model [12] uses a stack–walking algorithm for deciding whether the access to some resource should be permitted or not. Access is granted if *all* the code on the stack has the required permission (we ignore doPrivileged for simplicity, as it does not affect this issue). Since continuations capture (portions of) the stack, they carry around the security context of the code who captured them, and makes it more difficult to reason about security. Indeed, it's not only control flow (who calls whom), but also data flow (what continuations can reach a given `call`) that has to be taken into account. Imagine a continuation that points to some security sensitive operation (like formatting the hard drive) that reaches untrusted code (by careless programming). Untrusted code can call this continuation and no one could stop disaster from happening. This problem is similar to that of thread creation: a new thread has an empty stack, therefore it could perform some sensitive actions which, later, could leak to the code that created the thread. That code might not have had the necessary permissions to carry on those operations itself. The security model handles this by making new threads inherit the security context of their parent.

A similar solution for continuation would be to record for each thread the security contexts at all points where a continuation was called, and modify the algorithm to take them into account. However, this is not a satisfactory solution since it implies an ever increasing chain of security contexts. Our implementation does not address this issue, which should however be kept into consideration when dealing with security-sensitive applications.

## 4 Implementation

We implemented continuations in Ovm [13], a framework for building virtual machines, coming with several implementation of VM services (garbage collection, monitors, or execution engine) which can be combined to build a working JVM. We used the *j2c* execution engine, which is an ahead-of-time compiler that uses C++ as target.

### 4.1 Garbage Collection

We used a conservative, mostly copying garbage collector. Using C++ as a target has the disadvantage that the garbage collector has to be able to deal with lack

of precision. While pointers from objects can be precisely identified, pointers from the stack have to be handled conservatively, and the pointed objects have to be marked as not movable (pinned). Since continuations are not ordinary objects, but carry stacks with them, they need to be visited *before* traversing the reachability graph, and their "neighbours" pinned. This last requirement comes from the fact that an object can be reached before its continuation is visited, and therefore moved before it had the chance to be pinned. Note that this issue does not appear in VMs that use heap–allocated activation frames, such as [14], as they can be treated as ordinary objects.

We modified the mostly copying GC to handle continuations correctly. We keep around a list of *live* continuations which is updated each time a continuation is captured (continuation capture, as well as continuation calls are atomic operations). When the GC starts, it visits all live continuations and treats their stacks and registers as conservative roots. A subtle problem arises: since Ovm provides no weak references, and continuations are referenced from the GC itself, they will never be collected. Our solution is to "manually" garbage collect the live continuation list. We use a simple mark and sweep algorithm which marks continuations in the live list when they are visited during normal GC walk. At the end of the GC the list is swept. This way we guarantee that dead continuations will be collected in the second GC run after they are dead.

### 4.2 Monitors

In order to satisfy the monitor-affinity property of continuations, we need to keep track of the acquired locks. Each thread maintains a list of entered monitors, which is updated on monitor enter and exit. When a continuation is captured, the list of monitors is saved. When a continuation is called, we check that the monitors entered by the current thread are the same as the ones saved in the continuation. If this is not true, an `IllegalMonitorState` exception is thrown. Continuations are invalidated eagerly, as soon as the most recently acquired monitor has been released. This allows the programmer to test a continuation before attempting to call it.

A current limitation is that thin locks [15] are not supported. Thin locks use a partial word in objects to perform fast locking when there is no contention.

### 4.3 Performance

We present some preliminary performance results of our prototype. Figure 4 shows the cost of different operations involving continuations. Our testing configuration is real time Ovm with the j2c backend, and a mostly–copying garbage collector. Each data point in the figure is an average over ten measurements.

Continuation capture is the most expensive operation. Its cost increases lineary with the stack depth, and up to depths of 50 activation frames it is lower than the cost of creating a thread in our system. Its cost is roughly 8 times the cost of a normal call. One–shot continuations show a constant cost relative to stack depth, about the same as a normal method call. It is interesting to note

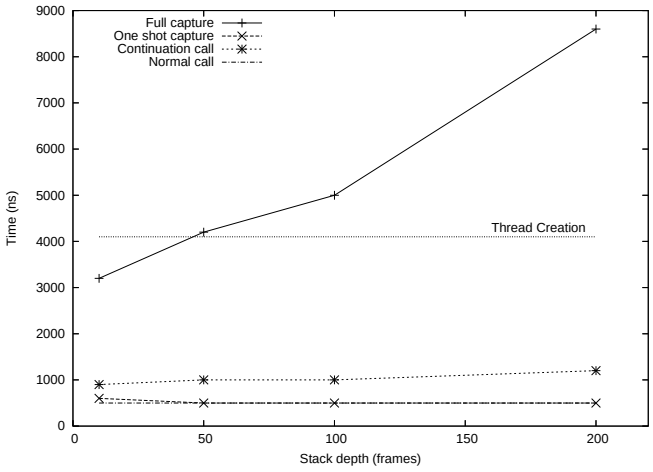


**Fig. 4.** Cost of continuation operations in different settings. We used a dual–core, Intel Xeon 3GHz machine with 2GB RAM, running Linux 2.6.15.

that the cost of a continuation call is almost constant relative to stack depth, roughly 2 times more expensive than method calls. We believe the difference between capture and call is due to memory allocation that takes place during capture. The cost of continuation capture is an area where we expect our future implementation to show significant improvement. We didn't include measurements for bounded continuations as their behavior relative to stack depth is similar to that of full continuations.

## 5 Conclusions

We have presented a way to integrate first–class continuations in the Java language. We studied the interactions between existing language features and continuations and suggested possible approaches to reconcile the existing semantics of Java regarding exceptions, threads, monitors and the security model. We have implemented continuations in a Java VM, we have a system that handles monitors correctly, and we conducted preliminary performance measurements. We showed how a conservative copying GC can be extended to deal with continuations. As far as we know, this is the first implementation of continuations in a Java VM.

## 6 Future Work

We are planning to conduct further work on the interaction between continuations and the "finally" exception mechanism, in order to obtain an efficient implementation that does not sacrifice the expected exception semantics.

Continuation capture is eagarly copying the whole execution stack. We plan to implement a more efficient scheme using lazy copying [16].

## References


1. John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6, 1993.
2. Paul Graunke, Shriram Krishnamurthi, Steve Van Der Hoeven, and Matthias Felleisen. Programming the Web with high-level programming languages. *Lecture Notes in Computer Science*, 2028:122, 2001.
3. Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside – a multiple control flow web application framework. *ESUG International Smalltalk Conference*, September 2004.
4. Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. Continuations from generalized stack inspection. In *International Conference on Functional Programming*, 2005.
5. Rife continuations. http://rifers.org/wiki/display/RIFECNT/Home.
6. Matthias Felleisen, Daniel P. Friedman, Bruce Duba, and John Merrill. Beyond continuations. Technical Report 216, Indiana University, 1987.
7. Carl Bruggeman, Oscal Waddell, and R. Kent. Dybvig. Representing control in the presence of one-shot continuations. In *Conference on Programming Language Design and Implementation*. ACM SIGPLAN, 1996.
8. Christian Queinnec. A library of high-level control operators. *Lisp Pointers, ACM SIGPLAN Special Interest Publ. on Lisp*, 6(4):11–26, 1993.
9. Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, April 1999.
10. R. Kelsey, W. Clinger, and J. Rees. *Revised$^5$ Report on the Algorithmic Language Scheme*, chapter 6.4. 1998.
11. Dorai Sitaram. Unwind-protect in portable scheme. In *Scheme Workshop 2003*, November 2003.
12. Li Gong. Java 2 security architecture. 1998.
13. The ovm project. http://ovmj.org.
14. John H. Reppy. A high-performance garbage collector for Standard ML. Technical report, Murray Hill, NJ, 1993.
15. David F. Bacon, Ravi B. Konuru, Chet Murthy, and Mauricio J. Serrano. Thin locks: Featherweight synchronization for java. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–268, 1998.
16. William D. Clinger. Implementation strategies for first–class continuations. *Higher–Order and Symbolic Computation*, 12:7–45, 1999.