# Exception Handling During Asynchronous Method Invocation

Aaron W. Keen and Ronald A. Olsson

Department of Computer Science, University of California, Davis, CA 95616 USA
{keen,olsson}@cs.ucdavis.edu

**Abstract.** Exception handling mechanisms provided by sequential programming languages rely upon the call stack for the propagation of exceptions. Unfortunately, this is inadequate for handling exceptions thrown from asynchronously invoked methods. For instance, the invoking method may no longer be executing when the asynchronously invoked method throws an exception. We address this problem by requiring the specification of handlers for exceptions thrown from asynchronously invoked methods. Our solution also supports handling exceptions thrown after an early reply from a method and handling exceptions after forwarding the responsibility to reply. The JR programming language supports the exception handling model discussed in this paper.

## 1 Introduction

Asynchronous method invocation facilitates the dynamic creation of concurrently executing threads and the communication between such threads. When a method is asynchronously invoked, the invoking thread continues to execute while another thread executes the body of the invoked method. Such concurrent execution can benefit both the design of and the performance of a program. The use of asynchronous method invocation, however, complicates the use of exceptions and, in particular, the handling of thrown exceptions.

Exception handling mechanisms for sequential programming languages are well-understood. Goodenough [2] presents a general discussion of the issues that exception handling mechanisms need to address and the different semantics (e.g., terminate, retry, or resume) provided by such mechanisms. Yemini and Berry [9] propose an additional model (replacement) that allows an erroneous subexpression (one that raises an exception) to be *replaced* by the handler's result.

The exception handling mechanisms provided by sequential programming languages rely upon the call stack for the propagation of exceptions. A thrown exception is propagated (either implicitly or explicitly) up the call stack until an appropriate handler is found. In Figure 1, method `baz` throws an exception. The exception is propagated through method `bar` and into method `foo`, where it is finally handled.

Unfortunately, the reliance of such mechanisms on the call stack is inadequate for handling exceptions thrown from an asynchronously invoked method.

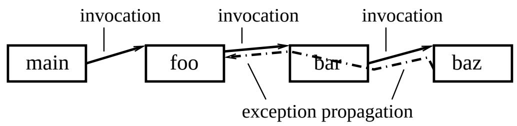


**Fig. 1.** Exception propagated through call stack.

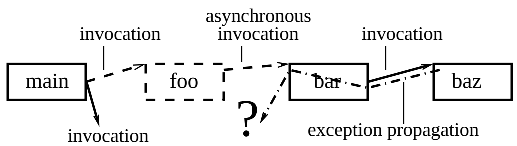


**Fig. 2.** Exception propagated from method invoked asynchronously.

Figure 2 depicts the same program as before, but with the method `bar` invoked asynchronously. Again, an exception is thrown from method `baz` and propagated to method `bar`. If method `bar` does not have an appropriate exception handler, then the exception must be further propagated. But, since method `bar` was invoked asynchronously, the preceding call stack is not accessible. In fact, the method that invoked `bar` (i.e., `foo`) may no longer be executing. As such, the call stack cannot be used to propagate exceptions from methods invoked asynchronously.

We address this problem by requiring the specification of handlers for exceptions thrown from methods invoked asynchronously. The exception handling support presented in this paper bears some resemblance to that provided in both ABCL/1 [3] and Arche [4], which are discussed in greater detail in Section 4. Our solution, however, differs in the support for static checks for handling exceptions, handling exceptions after an early reply from a method, and handling exceptions after forwarding within a method. Both early reply and forwarding are useful in distributed programming [7, 1]. Early reply allows the invoked method to transmit return values to its invoker, yet the invoked method continues to exist and executes concurrently with its invoker.[1] Forwarding passes the responsibility for replying to the invoker from the invoked method to another method. The exception handling support discussed in this paper has been designed and implemented as part of the JR programming language [6, 5].

## 2 Exceptions During Asynchronous Invocation

The JR programming language extends Java with support for, among other features, asynchronous method invocation via the `send` statement. To facilitate the handling of exceptions thrown from an asynchronously invoked method, we

[1] Further discussion of support for reply is discussed in [5].

```
public class IOHandler implements edu.ucdavis.jr.Handler {
    public handler void handleEOF(java.io.EOFException e)
    { /* handle exception */ }
    public handler void handleNotFound(java.io.FileNotFoundException e)
    { /* handle exception */ }
}
```

**Fig. 3.** Class definition for a simple *handler* object.

```
IOHandler iohandler = new IOHandler();
...
send readFile("/etc/passwd") handler iohandler;
...
```

**Fig. 4.** Specification of a *handler* object during a send.

require the specification of a *handler* object as part of a `send`. Any exceptions propagated out of the invoked method will be directed to the *handler* object. As such, in accordance with the verbose nature of Java's exception handling model, JR requires that the specified *handler* object is capable of handling any exception potentially thrown from the invoked method.

To be used as a *handler*, an object must implement the `Handler` interface and define a number of *handler* methods. A method is defined as a *handler* through the use of the `handler` modifier (much like the `public` modifier). A *handler* method takes only a single argument: a reference to an exception object. Each *handler* method specifies the exact exception type that it can handle. When an exception is delivered to a *handler* object, it is handled by the *handler* method of the appropriate type. If two *handler* methods may handle the exception, then the method handling the most specific type is selected. *Handler* methods cannot propagate exceptions out of their method body.

An example definition of a *handler* object's class is given in Figure 3. In this example, *handler* objects of type `IOHandler` can handle end-of-file and file-not-found exceptions. An exception of type `java.io.EOFException` directed to such a *handler* object will be handled by the `handleEOF` method.

A `send` statement must specify, using a `handler` clause, the *handler* object that is to be used to handle any exceptions propagated from the asynchronously invoked method. An example of the specification of a handler object is given in Figure 4. The JR compiler statically checks that the specified *handler* object can handle each of the potentially thrown exceptions.

## 3 Exceptions After Forwarding

A modification of the standard synchronous invocation semantics is forwarding. An invoked method may *forward* to another method the responsibility of replying. For example, in Figure 5, method `foo` does some calculations and then forwards responsibility to method `bar`, after which the two methods execute concurrently.

```
int baz(String filename) throws java.io.EOFException {
    int retval = foo(filename);
    // retval actually comes from bar because of foo's forward
    ...
}
int foo(String filename) throws java.io.EOFException {
    ...
    IOHandler iohandler = new IOHandler();
    forward bar(filename) handler iohandler;  // forward invocation
    ...  // continue executing
}
int bar(String filename) throws java.io.EOFException {
    ...  // potentially throw java.io.EOFException
}
```

**Fig. 5.** Forwarding.

A `forward` statement must specify a *handler* object to handle any exceptions thrown by the forwarding method after executing the `forward` statement. Any exceptions thrown prior to executing a `forward` statement are propagated according to the manner in which the method was invoked. Any exceptions thrown by the forwarding method after executing a `forward` statement are directed to the *handler* object. The method to which responsibility is forwarded inherits the handler of the forwarding method: the call stack link if invoked synchronously and a handler object if invoked asynchronously.

## 4 Discussion

Handler objects are implemented as *Serializable* Java objects. The *handler* methods are implemented as normal methods, but are gathered during compilation to generate a dispatch method (named by the *Handler* interface). The dispatch method is necessary as a single, well-named entry point into each *handler* object. All exceptions are directed to the dispatch method, which routes each exception to the appropriate *handler* method. A *handler* object is sent, as an additional parameter, to an invoked method, and used only when an exception is raised. As such, the *handler* object will exist during the duration of the method execution.

A previous approach [8] to handling exceptions thrown from asynchronously invoked methods allows for the specification of an exception handler when an exception is actually raised. Unfortunately, specifying the handler at the point an exception is raised introduces some limitations. For example, it might be desirable for a method that can be invoked both synchronously and asynchronously to propagate exceptions up the call stack or to a handler, as appropriate . Such a distinction would require support for a method to determine how it was invoked.

As mentioned previously, the solution proposed in this paper bears some resemblance to the solutions for ABCL/1 [3] and Arche [4]. ABCL/1 allows synchronous, asynchronous, and future-based method invocation. Each invocation may specify a "complaint" destination. Any exception raised during the execution of the method will be directed to the "complaint" object, if specified.

Otherwise, the exception is propagated to the invoker through the call stack. ABCL/1, due to the nature of the language, does not perform static checks on the "complaint" destination to ensure that it can handle the thrown exceptions.

The exception handling support in Arche is similar to that provided by ABCL/1. A set of handler objects can be specified as part of an asynchronous invocation. Exceptions thrown from the invoked method are directed to each of the specified handler objects. Arche also statically checks that each of the handler objects can actually handle the potentially thrown exceptions.

## 5 Conclusion

This paper presented the design and implementation of an exception model that supports handling exceptions thrown from an asynchronously invoked method, handling exceptions thrown after an early reply from a method, and handling exceptions after forwarding. The JR programming language supports *handler* objects and the presented exception model.

## References


1. G.R. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1991.
2. J. B. Goodenough. Exception handling issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, 1975.
3. Y. Ichisugi and A. Yonezawa. Exception handling and real time features in an object-oriented concurrent language. In *Proceedings of the UK/Japan Workshop on Concurrency: Theory, Language, and Architecture*, pages 604–615, 1990.
4. V. Issarny. An exception handling mechanism for parallel object-oriented programming: toward reusable, robust distributed software. *Journal of Object-Oriented Programming*, 6(6):29–40, 1993.
5. A. W. Keen. *Integrating Concurrency Constructs with Object-Oriented Programming Languages: A Case Study.* PhD dissertation, University of California, Davis, Department of Computer Science, June 2002.
6. A. W. Keen, T. Ge, J. T. Maris, and R. A. Olsson. JR: Flexible distributed programming in an extended Java. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS 2001)*, pages 575–584, April 2001.
7. B. Liskov, M. Herlihy, and L. Gilbert. Limitations of remote procedure call and static process structure for distributed computing. In *Proceedings of 13th ACM Symposium on Principles of Programming Languages*, St. Petersburg, FL, January 1986.
8. A. Szalas and D. Szczepanska. Exception handling in parallel computations. *ACM SIGPLAN Notices*, 20(10):95–104, October 1985.
9. S. Yemini and D. M. Berry. A modular verifiable exception handling mechanism. *ACM Transactions on Programming Languages and Systems*, 7(2):214–243, 1985.