A Side-by-Side Comparison of Exception Handling in Ada and Java

Alfred Strohmeier, in collaboration with Stanislav Chachkov

Swiss Federal Institute of Technology in Lausanne (EPFL)
Software Engineering Laboratory
CH-1015 Lausanne EPFL, Switzerland
alfred.strohmeier@epfl.ch

1 Introduction

The purpose of this paper is to compare the exception handling mechanisms of Ada and Java. In order to be intelligible and useful to both communities, we have tried not to get into specific technical intricacies of the languages, perhaps sometimes at the cost of precision. Nevertheless, we decided to use the language-specific terminology whenever we write about a given language. We believe that the contrary would often lead to misunderstandings: a) the same term sometimes covers two different concepts, e.g. object, or b) when the concepts are basically the same, the features provided by the language can be largely different, making any unification impossible, e.g. task in Ada and thread in Java.

Table 1 “Comparison of Terminology” shows the meanings and correspondences of the most important terms related directly or indirectly to exception handling in the two languages.

We did not try to assess the merits of the two languages, not even with regard to exception handling. The issues are too many, and would by far exceed the scope of this paper: How to assess trade-offs between expressive power and performance, if ever a definite evaluation of performance is possible? How to assess language constructs whose “good” intent cannot be enforced? E.g. in Java, the “lazy” programmer can declare Throwable in a method’s throws clause, instead of referring to specific exception classes. Where is the borderline between responsibility left to the programmer, and an admittedly error-prone construct in the language?

Our goal was therefore to present facts, and facts only, as much as this is possible without any interpretation.

2 Overview of the Contents

Whenever possible, we tried to keep the presentations of the two languages in parallel, at least at the section level:

- Language Summary - Language Summary
- Exception Names and Exception Occurrences - Exception Classes and Instances
- Predefined Exceptions - Predefined Exception Classes
- Raising an Exception - Throwing an Exception
- Language-defined Checks - Language-defined Checks

- Handling Exceptions - Catching Exceptions
- Visibility of Exceptions - Checked and Unchecked Exceptions
- Carrying Information with an Exception - Carrying Information with an Exception
- Performance Issues - Performance Issues
- Concurrent Programming - Concurrent Programming
- Generics

3 Acknowledgement

We would like to thank Ben Brosgol and the anonymous reviewer for their very valuable and detailed comments.

4 References

The main reference for the Ada language is the International ISO Standard; we will refer to it by the acronym RM [1]. For the Java language, our work is based on [2], referred to as JLS.

[1] S. Tucker Taft, Robert A. Duff (Eds.); Ada 95 Reference Manual: Language and Standard Libraries, International Standard ISO/IEC 8652:1995(E); Lecture Notes in Computer Science, vol. 1246; Springer-Verlag, 1997; ISBN 3-540-63144-5.

[2] James Gosling, Bill Joy, Guy Steele, Gilad Bracha; The Java Language Specification (Second Edition); Addison-Wesley, 2000; ISBN 0-201-31008-2

Ada		Java	
Term	Meaning	Term	Meaning
type	data type or class or task type	class	class
object	variable or constant	object	class instance or array
operation	subprogram	method	
exception [name]	a name for an exception type	exception class	the class Throwable or one of its subclasses
exception occurrence	a dynamically created exception instance	exception	an instance of an exception class
		checked exception	the compiler checks that the program contains a handler
		unchecked exception	belong to the classes Error or RuntimeException or their subclasses
predefined exceptions	Constraint_Error, Storage_Error, Program_Error, Tasking_Error	exception classes defined in java.lang	these exception classes can be referred to by simple names
raise an exception		throw an exception	
handle an exception		catch an exception	
handled sequence of statements		try statement	
exception handler		catch clause or handler	
task		thread	
task activation		starting a thread	
protected object	all operations of a protected object are performed in mutual exclusion	object's monitor	a locking mechanism is associated with any object for monitoring concurrent access
protected operation	an operation of a protected object	synchronized statement or method	a block or method executed in mutual exclusion

Table 1: Comparison of Terminology

Ada

1 Language Summary

1.1 . Ada is a block-structured language with nesting blocks (like Pascal). In a block, declarations precede the statements. Subprograms and block statements are typical blocks.

1.2 . The module construct is called a package. A package can contain/define types, subprograms and exceptions, among others. Packages can be organized in hierarchies. A hierarchy of packages forms a name space.

1.3 . A package can declare names for exceptions.

1.4 . A subprogram cannot declare in its signature the exceptions it might raise/ propagate.

1.5 . An exception might be propagated to a place where its name is not visible. It is possible to handle such an “anonymous” exception and then raise it again.

1.6 . Exceptions are a special construct in the language, not related to other constructs. An exception cannot be a component of a composite type or a parameter. There is however an ad hoc construct as a work around (see Identity attribute).

1.7 . The language provides extensive support for concurrent programming (tasks and protected objects).

2 Exception Names and Exception Occurrences

2.1 Terminology. An exception (occurrence) is said to be raised at the place

Java

1 Language Summary

1.1 . Java is a block-structured language, except that methods cannot be (directly) declared locally. Local variables, e.g. class instances, can be declared within a statement sequence. A method cannot be nested inside another method.

1.2 . A package defines a name space. It can contain/define class and interface declarations. The interface and class constructs are the module constructs of the language. Classes can be nested.

1.3 . Exception classes are declared and instantiated in the same way as other classes.

1.4 . A method must declare in the throws clause of its signature all exceptions it might throw or propagate (let go unhandled), except exceptions of the classes Error and RuntimeException.

1.5 . At least a superclass of the exception class is visible at all places where the exception can be propagated to.

1.6 . An exception is an ordinary object. It can be referenced in the field of another object or passed as a parameter.

1.7 . The language provides built-in support for dealing with concurrent execution of a region of code (synchronized statements and methods) and for thread coordination (wait/notify/notifyAll methods of the class Object). Additional support for concurrent programming is mainly by means of libraries, especially the class Thread.

2 Exception Classes and Instances

2.1 Terminology. An exception is said to be thrown from the point where it occurred where it occurred and is said to be handled at the point to which control is transferred. When an exception is not handled and control is transferred, the exception is said to be propagated.

2.2 Model. The Ada reference manual does not relate the concept of an exception to other language constructs. However, we think the following model is mostly accurate:

There is so to speak a single predefined abstract limited datatype denoted by the keyword exception. A predefined or user-defined exception is a concrete direct subtype of this abstract datatype. An exception occurrence is a dynamically created object of this datatype.

However, this is only a mental model, and the datatype is not recognized as such by the language. Therefore, it is not possible to declare and name an exception occurrence, and exceptions cannot be formal or actual parameters of subprograms, or components of composite types (like arrays or records).

2.3 Exception Declaration. To stick to the RM, "[...] an exception declaration declares a name for an exception.", for example:

```
    Transmission_Error: exception;
```

An exception can be declared wherever a declaration can occur, e.g. in the declarative parts of a block statement, a subprogram, a package, etc.

and is said to be caught at the point to which control is transferred.

2.2 Model. Every exception is represented by an instance of the class Throwable or one of its subclasses. Throwable is directly derived from Object, the "mother" of all classes:

```
public class Throwable {
    public Throwable() {...}
    public Throwable(String message) {...}
    public String toString() {...}
    public String getMessage() {...}
    ...
}
```

Throwable has two subclasses, named Error and Exception. For an application programmer, the usual approach is to define subclasses of Exception. Whenever needed, s/he then throws an exception of one of these subclasses. Typically, the instance is freshly created in the context of the exceptional situation (in order to include accurate information, such as stack trace data).

The class Throwable and its subclasses are called exception classes.

Because Java exception classes are subclasses of Object, they can be method parameters, components of composite types etc.

2.3 Exception Declaration. The programmer declares an exception by declaring an exception class, most of the time a subclass of Exception:

```
    public class TransmissionError
        extends Exception {...}
```

An exception class can be declared wherever a class can be declared.

As expected, it is possible to overload constructors and override inherited methods, e.g.

```
class BlewIt extends Exception {
    BlewIt() {}
    BlewIt(String s) {super(s);}
}
```

3 Predefined Exceptions

There are four pre-defined exceptions:

Constraint_Error,

Program_Error,

Storage_Error,

Tasking_Error.

3 Predefined Exception Classes

There is a large number of exception classes predefined in the package java.lang. The top-levels of the inheritance tree are shown in figure 1:

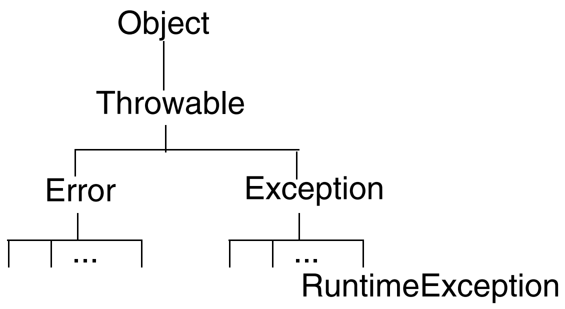

Figure 1: Hierarchy of Predefined Classes

4 Raising an Exception

4.1 Principle. As stated by the RM: “An exception is raised initially either by a raise statement or by the failure of a language-defined check.” Such a failure is detected by the execution support during run-time.

4.2 Raise Statement. A raise statement can occur at any place where a statement can occur, and has a very simple syntax:

```
raise exception_name;
```

Example of raising an exception:

```
raise Transmission_Error;
```

4.3 Language-Defined Check. The failure of a language-defined check raises one of the four predefined exceptions.

4 Throwing an Exception

4.1 Principle. As stated by the JLS: “When a Java program violates the semantic constraints of the Java language, a Java Virtual Machine signals this error to the program as an exception. [...] Java programs can also throw exceptions explicitly, using throw statements.”

4.2 Throw Statement. A throw statement can occur at any place where a statement can occur, and has a very simple syntax:

```
throw Expression;
```

where the type of Expression must be a Throwable or a subclass thereof.

Example of throwing an exception:

```
throw new TransmissionError();
```

or, but unusual:

```
TransmissionError myException
    = new TransmissionError();
throw myException;
```

4.3 Language-Defined Check. The failure of a language-defined check throws an exception of one of the predefined subclasses of Error or RuntimeException.

5 Language-defined Checks

5.1 Constraint_Error. By far the most often raised predefined exception is Constraint_Error. It is raised upon an attempt to violate a constraint imposed on a value, e.g. a range constraint, an index constraint, a discriminant constraint, or upon an attempt to use a record component that does not exist, to use an indexed component that does not exist, to access an object by a pointer that is null, or upon an attempt to divide by zero, etc.

5.2 Storage_Error. Generally speaking, the exception Storage_Error is raised when there is no more dynamic memory available: the evaluation of an allocator requires more space than is available (in "its" storage pool), or the space available for a task or a subprogram has been exceeded.

5.3 Program_Error. The exception Program_Error is raised in cases when the program is not "well-formed", but the error cannot be detected during compilation, e.g. because the problem is undecidable. E.g. Program_Error is raised upon an attempt to call a subprogram whose body has not yet been elaborated, i.e. has not yet been "created".

5.4 Tasking_Error . Generally speaking, the exception Tasking_Error is raised whenever there is a communication problem between tasks. E.g. it is raised if the activation of a task fails, or if an attempt is made to call an entry of a task that has already completed its execution, or if an attempt is made to retrieve the priority of a completed task, etc.

5 Language-defined Checks

5.1 RuntimeException and Error. Most of the time, a predefined exception belongs to a subclass of RuntimeException, e.g. ArithmeticException, ArrayStoreException, ClassCastException, IllegalArgumentException, IndexOutOfBoundsException, NullPointerException, etc. It might be noticed that there are no range checks in Java, and an integer arithmetic overflow simply causes "wrap around".

Ordinary programs are usually not expected to recover from exceptions in the class Error and its subclasses. Error has the subclasses LinkageError, VirtualMachineError and ThreadDeath.

5.2 VirtualMachineError: Lack of Memory. The errors OutOfMemoryError and StackOverflowError are thrown by the Java Virtual Machine when there is a lack of memory to continue execution. Other subclasses of VirtualMachineError are used to signal an internal error.

5.3 LinkageError: Consistency of Program. Because of the highly dynamic nature of Java program composition, errors and exceptions due to inconsistencies are especially important. Errors detected when a loading, linkage, preparation, verification or initialization failure occurs are reported by throwing an exception of the subclass LinkageError, e.g. ClassFormatError, ClassCircularityError, InstantiationError, NoSuchFieldError, NoSuchMethodError, VerifyError, ExceptionInInitializerError, etc.

5.4 Thread-Related Exceptions. The IllegalMonitorStateException is thrown when a thread tries to perform an operation (wait or

6 Handling Exceptions

6.1 Handled Sequence of Statements.

The basic idea in Ada is to associate an exception handling part with a sequence of statements. The exception handling part deals with exceptions that arise from the execution of the sequence.

The following constructs, each comprising a sequence of statements, can contain an exception handling part:

- a block statement,

- a subprogram body,

- a package body,

- a task body,

- an accept statement,

- an entry body.

In Ada, a block statement consists in a leading declarative part, a sequence of statements, followed by an exception handling part. In block statements and subprogram bodies, the exception handling part deals with failures and exceptions raised during the execution of the block or the subprogram. The exception handling part of a package body is there to handle problems that arise during its "creation", i.e. elaboration of the package. Exception handling parts in accept statements and entry bodies are used to deal with communica-

notify) on an object whose lock it did not previously acquire.

The IllegalThreadStateException is thrown to indicate that a thread is not in an appropriate state for the requested operation.

If a thread t is interrupted (via a call of t.interrupt()) then an InterruptedException is thrown in t either immediately (if t is blocked on a call of wait(), join(), or sleep()) or when it next blocks on one of these calls.

6 Catching Exceptions

6.1 Try Statement. The only place to catch an exception is in a try statement. A try statement executes a block. In Java, a block consists in a sequence of statements and variable declarations. The catch clauses at the end following the block can handle exceptions thrown during the execution of the block.

6.2 Syntax.

```
TryStatement:
    try Block Catches
    try Block Catches_opt Finally
Catches:
    CatchClause
    Catches CatchClause
CatchClause:
    catch (FormalParameter) Block
Finally:
    finally Block
```

6.3 Example.

```
try {
    int a[] = new int[2];
    a[4];
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("exception: "
                        + e.getMessage());
    e.printStackTrace();
}
```

6.4 Dynamic Semantics. If an exception is thrown during the execution of the block and there is a catch clause that can catch it, then control will be transferred to the first tion problems between tasks.

6.2 Syntax. The form of an exception handling part is similar to a case statement, each case containing one exception handler dealing with one or several exceptions. A last "others" case can handle all the exceptions not named in previous cases:

```
handled_sequence_of_statements ::=
    sequence_of_statements
    [exception_handling_part]
exception_handling_part ::=
    exception
        exception_handler
        {exception_handler}
exception_handler ::=
    when [choice_parameter_specification:]
        exception_choice {| exception_choice} =>
            sequence_of_statements
choice_parameter_specification ::=
    defining_identifier
exception_choice ::= exception_name | others
```

6.3 Example.

```
begin
    ... -- call operations in File_System
exception
    when End_Of_File =>
        Close (Some_File);
    when File_Not_Found =>
        Put_Line ("Some specific message"));
    when others =>
        Put_Line ("Unknown Error");
end;
```

6.4 Dynamic Semantics. When an exception arises, control is transferred to the user-provided exception handler at the end of the sequence of statements where the exception arises, if there is an exception handling part, and if it contains a handler for the raised exception, or it is propagated to the dynamically enclosing execution context. In both cases, the sequence of statements where the exception arose is abandoned, and it is impossible to return to the offending statement. These semantics correspond to the so-called "Termination Model". Propagating the exception means that it is reraised at the place where the current sequence of statements was such catch clause.

A catch clause can catch an exception if the run-time type of the exception can be assigned to the FormalParameter of the catch clause, i.e. the FormalParameter belongs to the same class or to a superclass of the thrown exception. Several catch clauses can fulfil this requirement, and there is therefore a rule that the first one is chosen. As a result, the programmer must write the catch clauses by starting with the most specific ones.

We will first deal with the case where there is a catch clause, but no **finally** block.

In this case, upon transfer of control to a catch clause, the value of the exception is assigned to the FormalParameter, and the block of the catch clause is executed. If that block completes normally, then the try statement completes normally. If that block completes abruptly for any reason, then the try statement completes abruptly for the same reason.

If none of the catch clauses can handle the exception, then the try statement completes abruptly because of the throw of the exception that was not handled.

If the try statement has a **finally** block, its block of code is executed, no matter whether the try block completes normally or abruptly, and no matter whether a catch clause is first given control. If the finally block terminates abruptly for reason S, then the try statement completes abruptly for the same reason. Note that a non handled exception in the try block or in a catch block is discarded in that case. If the finally block completes normally, then the try statement completes normally if there is no pending exception. It completes abruptly for reason V or R respectively if an exception V was thrown in the try block and not handled, or if the handling catch clause was abruptly terminated for reason R.

dynamically entered. E.g. when an exception is raised during the execution of a subprogram and not locally handled, it is reraised at the place of the subprogram call, and therefore propagated to the calling context.

Table 2 summarizes all situations.

try block: terminates	catch clause: exists, terminates	finally block	outcome
normally		normally	normally
		abruptly S	abruptly S
abruptly V	yes, normally	normally	normally
		abruptly S	abruptly S
	yes, abruptly R	normally	abruptly R
	no	normally	abruptly V
		abruptly S	abruptly S

Table 2: Outcome of Try Statement with a “finally” Block

7 Visibility of Exceptions

First of all, it must be noted that, to the contrary of Java, it is not required and not possible to declare in the signature of a subprogram the exceptions it might raise. The programmer who writes a call must therefore rely on comments supplied with the subprogram to know if it might raise an exception.

Secondly, and there is clearly a relationship with the first note, an exception can be propagated to an execution context where its name is not visible. Such an “anonymous” exception can still be handled by providing an exception handler with an “others” choice. It can even be reraised inside the exception handler by a raise statement mentioning no exception name. In order to distinguish between such “anonymous” exceptions, the programmer can associate locally a name with the specific exception occurrence, and then use the features provided by the package Ada.Exceptions (see 8.2) to handle it in some specific appropriate way:

7 Checked and Unchecked Exceptions

7.1 Throws Clause. In the declaration of a method or constructor, all checked exceptions it might propagate (let go unhandled) must be declared in a throws clause at the end of the header. The syntax for this part of the header is:

```
Throws:
    throws ClassTypeList
ClassTypeList:
    ClassType
    ClassTypeList, ClassType
Example:
void blowUp() throws BlewIt {
    throw new BlewIt();
}
```

7.2 Checked and Unchecked Exceptions. All subclasses of Error and of RuntimeException are unchecked exception classes, and a method is not required to declare them in its throws clause even if an instance might be thrown during its execution but not caught. Exceptions of the class Error should never occur and ordinary programs are not expected to recover from

```
exception
    when Error: others =>
        ...
        ... Ada.Exceptions.
            Exception_Information (Error)...
end;
```

8 Carrying Information with an Exception

8.1 Name of an exception and message.

It is possible to retrieve the name of an exception at run-time (see RM 11.4.1 (12)), providing therefore a form of reflection for programming exception handling.

Also, implementation-defined information and a message, each a character string, are associated dynamically with every exception occurrence. The contents of the message can be supplied by the programmer when raising the exception. Both can be accessed by the programmer when handling the exception.

The mechanism used for these features is not really well integrated with the language, due to the special nature of exceptions, but is quite clean.

8.2 Package Ada.Exceptions. There is a unique identity associated with each exception, and each dynamic occurrence of an exception defines an exception occurrence. Both the identity of an excep-

them. Exceptions from the class RuntimeException and its subclasses are so frequent, that the language designers decided that it would be cumbersome for the programmer to declare them all in the method header.

All other Throwable subclasses are checked, i.e. the compiler checks the following rule. If an exception of a checked subclass can be thrown during the execution of a constructor or method without being caught, then itself, or one of its superclasses, must be declared in the throws clause of the header. The permission to declare a superclass can be used to realize a hierarchical exception handling architecture, but it can also be misused by the “lazy” programmer.

8 Carrying Information with an Exception

8.1 Message. As shown by its declaration, class Throwable and its subclasses have two constructors, one that takes no arguments and one that takes a String argument that can be used to produce an error message.

8.2 StackTrace. In fact, each Throwable contains also a stack trace, i.e. the trace of method calls up to the point where it was thrown.

8.3 Example.

```
try {
    int a[] = new int[2];
    a[4];
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println
        (“exception:” + e.getMessage());
    e.printStackTrace();
}
```

8.4 Additional Information. In addition, as exceptions are normal objects, the programmer can attach to them any needed tion and the exception occurrence are "regular" datatypes, which can be used as subprogram parameters. The package Ada.Exceptions defines these datatypes and the applicable subprograms.

To associate a user-defined message with an exception when raising it, it must be raised by calling the procedure Raise_Exception of package Ada.Exceptions, instead of using the raise statement:

Example:

```
Raise_Exception
    (Transmission_Error'Identity,
    "Time-out occurred.");
```

8.3 Naming an Exception Occurrence in a Handler. For accessing the name of an exception or the message it carries, the mechanism used is to declare locally a name for the exception occurrence when handling the exception. For instance, when handling the exception raised in the example of 8.2, the following handler will print the full name of the exception ending with its simple name, i.e. "Transmission_Error", followed by "Time-out occurred.".

Example:

```
exception
    when Stalled: Transmission_Error =>
        Print (Exception_Name (Stalled) &
             Exception_Message (Stalled));
```

information when defining an exception class.

9 Performance Issues

9.1 Suppressing Checks. By using the pragma Suppress, i.e. a directive to the compiler, permission can be given to an implementation to omit certain language-defined checks.

9.2 Exceptions and Optimization. In order not to impact negatively on the performance of some kinds of computing systems, e.g. superscalar processors, permission is given to an implementation not to raise an exception due to a lan-

9 Performance Issues

9.1 Suppressing Checks. Language-defined checks are always performed, and cannot be suppressed.

9.2 Exceptions are precise. The JLS states in 11.3.1: "Exceptions are precise: when the transfer of control takes place, all effects of the statements executed and expressions evaluated before the point from which the exception is thrown must appear to have taken place. No expressions, statements, or parts thereof that guage-defined check at the exact place where it occurred in a sequence of statements.

occur after the point from which the exception is thrown may appear to have been evaluated. If optimized code has speculatively executed some of the expressions or statements which follow the point at which the exception occurs, such code must be prepared to hide this speculative execution from the user-visible state of the program."

10 Concurrent Programming

10.1 Concurrent Constructs. The concurrent constructs in Ada are tasks and protected objects. A task is a lightweight process and corresponds to a thread in other languages. A protected object is like a monitor, with possible barrier conditions. It provides access in mutual exclusion to some internal data structure. Since a protected object is a passive entity, handling of exceptions does not lead to any specific problems. With regard to exception handling, protected operations are not different from normal operations.

As we will see, the predefined exception Tasking_Error is related to general communication failures between tasks.

10.2 Activation of a Task. The execution of a task begins with an activation phase. During this activation phase, entities local to the task are created; technically speaking, the declarations of the task body are elaborated. If an exception is raised during that activation phase, then the activation fails and the task becomes completed (completes its execution), and the predefined exception Tasking_Error is raised in the task that created the new task and initiated its activation.

10.3 Synchronization between Tasks.

Direct synchronization between tasks is performed through a rendezvous. One task, the client task, calls an entry of another task, the server task. The server

10 Concurrent Programming

10.1 Concurrent Constructs. The concurrent constructs in Java are threads and synchronized statements (including synchronized methods). A thread is a lightweight process and corresponds to a task in Ada. Threads never interact directly. Synchronized statements are used to arrange mutually exclusive access to a shared object across several threads: only one thread at a time is allowed to execute the block of statements of the synchronized statement. A synchronized statement therefore protects a region of code, and is close to a critical section bracketed by explicit acquire and release semaphore operations. The behavior of synchronized statements is described in the JLS in terms of locks. There is a lock associated with each object, i.e. with each instance of the class Object, but the programmer does not handle locks directly. The syntax of the synchronized statement is the following:

synchronized (Expression) Block

The Expression must be a reference type. Before starting execution of the Block, the current thread locks the lock associated with the referenced object. At completion of the Block, be it normally or abruptly, the lock is unlocked.

A single thread is permitted to lock a lock more than once.

A synchronized method is a notational convenience for enclosing the method body in a synchronized statement. For a class task serves such an entry call by executing an accept statement. If the server task is ready for serving the call, then the rendezvous takes place immediately. Otherwise, the calling task is queued, and will have to wait for the rendezvous to take place. During the rendezvous, information can be interchanged between the two tasks by passing parameters.

If an exception is raised during a rendezvous, i.e. the service to be provided by the called task ends by a not handled exception (i.e. the accept statement raises an exception and does not handle it), then it is propagated into both tasks as the same exception. This mechanism can be used by a server task to inform a caller task about some event.

During a rendezvous, if the called task is aborted, then the exception Tasking_Error is raised in the calling task. On the other side, if the calling task is aborted, the called task is not affected, and the rendezvous will be completed in a somewhat unusual way.

Tasking_Error is also raised in all tasks waiting on an entry when a task completes. Calling an entry of a task that is already completed also raises Tasking_Error in the caller.

10.4 Non Handled Exception in a Task. If an exception is not handled by a task at all, then, like the main program, the task is immediately completed and the exception is lost; it is not propagated to the parent unit. Tasks die therefore silently.

10.5 Syntax.

```
task_body ::=
    task body defining_identifier is
        declarative_part
    begin
        handled_sequence_of_statements
    end [task_identifier];
accept_statement ::=
```

(static) method, the lock associated with the Class object is used. For an instance method, the lock associated with “this”, i.e. the current instance, is used.

Usually, all methods of a class designed for concurrent use will be synchronized. The approach results in monitor-like object instances, since all method calls to an object instance are synchronized on this same instance. Note that the approach leaves ample room for programming errors, since the approach does not force all methods to be synchronized, even if they need to be protected.

10.2 Starting a Thread. A thread is created by creating an object of the built-in class Thread. The thread begins to run when its start method is called. If the start method is called for a running thread, it throws the exception IllegalThreadStateException.

10.3 Communication between Threads.

Communication between threads is always performed by means of shared objects. The language rules guarantee that proper use of synchronization constructs result in reliable transmission of values or sets of values from one thread to another through shared objects.

10.4 Thread Coordination. The methods wait, notify and notifyAll of class Object, and the methods join, interrupt, yield and sleep of the class Thread support coordination between threads (one could say “synchronization”, but this might cause confusion with synchronized statements). Instead of spinning, by locking and unlocking repeatedly an object, a thread can suspend itself by using wait until another thread awakens it using notify or notifyAll.

The behavior can be explained by so-called wait sets.

Every object, in addition to having an asso-

```
    accept entry_direct_name [(entry_index)]
                                  parameter_profile
        [do handled_sequence_of_statements
    end [entry_identifier]];

entry_index ::= expression

entry_body ::=
    entry defining_identifier
        entry_body_formal_part entry_barrier is
        declarative_part
    begin
        handled_sequence_of_statements
    end [entry_identifier];
```

ciated lock, has an associated wait set, which is a set of threads. When an object is first created, its wait set is empty. If a thread wants to call the method wait on an object, it must first acquire a lock on that object. Otherwise, wait will throw an IllegalMonitorStateException. The effect of calling the wait method on a synchronized object will result in placing the current thread in the object's wait set after the thread has released all locks on the object. Another tread can then awaken such a dormant thread by first acquiring a lock on the object owner of the wait set and calling then the notify or notifyAll method on it.

If a waiting thread is interrupted by another thread, then the wait method completes abruptly by throwing InterruptedException.

10.5 Non Handled Exception in a Thread. If an exception is thrown in a thread, and no handler is found for that exception, then the method uncaughtException is invoked for the ThreadGroup that is the parent of the current thread. Every effort is therefore made to avoid letting an exception go unnoticed.

As usual, a subclass of ThreadGroup can override the uncaughtException method, and the programmer can therefore take care of dealing with exceptions not handled in a thread, or more precisely, in any thread of a given thread group.

11 Generics

Ada has a powerful concept of generic units, i.e. generic subprograms and generic packages. A generic unit is a template of an ordinary unit, either a package or a subprogram. The instantiation of a generic unit will yield an ordinary unit, i.e. either a subprogram or a package. Instantiation can be thought of as a kind of macro-expansion. The formal parameter of a generic unit can be an object (a constant or variable), a type, a subprogram, or a package. Note

11 Generics

There are no generic classes in the current version of Java.

that it cannot be an exception.

For all entities declared within a generic unit, each instantiation will get its "own copies", and this also holds for exceptions. As stated by the RM: "If a generic unit includes an exception_declaration, the exception_declarations implicitly generated by different instantiations of the generic unit refer to distinct exceptions (but all have the same defining_identifier)."