Exceptions

The Java programming language provides a mechanism known as **exceptions** to help programs report and handle errors:

- When an error occurs, the program throws an exception.
- The normal flow of the program is interrupted and the runtime environment attempts to find an **exception handler**, a block of code that can handle a particular type of error.

Exception – an event that occurs during the execution of a program that disrupts the normal flow of instructions.

An object containing an information about the event is also called an exception.

Exceptions - Example

An example of usage of exceptions:

```
public static void main(String[] args) {
    try {
        int c = Integer.parseInt(args[0]);
        while (c-- > 0)  System.out.println(args[1]);
    }
    catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("Missing argument");
    }
    catch (NumberFormatException e) {
        System.err.println("\"" + args[0] +
                "\" isn’t an integer");
    }
}
```

```
Usage: java Example 3 xyz
```

Exceptions - Motivation

Let us consider a method that reads an entire file into memory. In pseudo-code it looks like this:

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

- What happens if the file can't be opened?
- What happens if the length of the file can't be determined?
- What happens if enough memory can't be allocated?
- What happens if the read fails?
- What happens if the file can't be closed?

Exceptions - Motivation (cont.)

```
errorCodeType readFile {
    initialize errorCode = 0;
    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
            . . .
```

Exceptions - Motivation (cont.)

```
            . . .
          close the file;
          if (theFileDidntClose && errorCode == 0) {
              errorCode = -4;
          } else {
              errorCode = errorCode and -4;
          }
      } else {
          errorCode = -5;
      }
      return errorCode;
}
```

Exceptions - Motivation (cont.)

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

Exception Objects

When an error occurs in Java:

- An exception object is created, it contains information about the exception (its type, the state of the program when the error occurred, ...).
- Normal flow of instructions is disrupted.
- The runtime system finds some code to handle the error.

An exception object is always an instance of some subclass of the class **java.lang.Throwable**. There are many standard exception classes and it is possible to define own exception classes.

Creating an exception and handing it to the runtime system is called **throwing an exception**.

The code that handles the exception is called an **exception handler**. The exception handler is said to **catch the exception**.

Which exception handler is chosen depends on the type of the exception object.

Catching Exceptions

There are three main components of a code that catches exceptions:

- the **try** block
- the **catch** blocks
- the **finally** block

The syntax is:

```
try  {
        . . .
} catch (. . .) {
        . . .
} catch (. . .) {
        . . .
} finally {
        . . .
}
```

A **try** block **must** be accompanied by at least one **catch** block or one **finally** block.

The `try` Block

In general a **try** block looks like this:

```
try {
    . . .    // Java statements
}
```

A **try** block is said to **govern** the statements enclosed within it and defines the scope of any exception handlers.

If an exception occurs within the **try** statement, that exception is handled by the appropriate exception handler associated with this **try** statement.

There can be any number of the **catch** blocks, but at most one **finally** block.

The `catch` Block(s)

The general form of a **catch** block is:

```
catch (SomeThrowableObject variableName) {
    . . .   // Java statements
}
```

A class SomeThrowableObject is a subclass of **java.lang.Throwable**. It declares the type of exceptions the handler can handle.

The variable variableName is the name by which the handler can refer to the exception.

This is a declaration of a local variable variableName. The scope of this variable is the body of the **catch** block.

The variable variableName can be used as any other local variable:

```
variableName.getMessage();
```

Note: The conventional name used for these types of variables is e.

The `catch` Block(s) (cont.)

The **catch** block contains a series of statements that are executed when the exception handler is invoked:

- If no exception occurs in the **try** block, all its **catch** blocks are skipped and the execution continues after them.
- If an exception of type T occurs in the **try** block and there is a **catch** block handling exceptions of type T (or its superclass), then this block is executed.

  If there is more than one handler that handles exceptions of type T then the first one matching handler is used.
- If there is no such handler, the runtime system looks for some other enclosing **try** statement and its handlers.

Note: Exceptions can be thrown everywhere, even inside the **catch** blocks.

The `catch` Block(s) (cont.)

The typical use of exception handlers:

```
try {
        . . .
} catch (ArithmeticException e) {
    System.out.println("Caught ArithmeticException: " +
                              e.getMessage());
} catch (IOException e) {
    System.out.println("Caught IOException: " +
                              e.getMessage());
}
```

The `finally` Block

The **`finally`** block provides a mechanism that allows to clean up the state of a method **regardless** of what happens within the **`try`** block.

Statements in the **`finally`** block are performed after:

- the **`try`** block exited normally,
- an exception occurred in the **`try`** block and was caught by some exception handler,
- an exception occurred in the **`try`** block and was not caught.

```
try {
        . . .  // opens a file and writes to it
} finally {
    if (file != null) {
        file.close();
    }
}
```

Exceptions and Methods

A method need not catch all exceptions, it can also throw exceptions to its caller.

If an exception of type T can occur in a method and the method does not catch the exception of type T, then we must specify that the method can throw an exception of type T.

To specify this, we add a **throws** clause to the header of the method:

```
public void readFile(String filename) throws IOException
{
        . . .
}
```

If a method can throw more than one type of exception we must specify all of them:

```
public Connection openConnection(Address addr)
    throws ConnectException, UnknownAddrException {
        . . .
```

Exceptions and Methods (cont.)

Any exception that can be thrown by a method is part of the method's public programming interface: callers of a method must know about the exceptions that a method can throw to intelligently and consciously what to do about those exceptions.

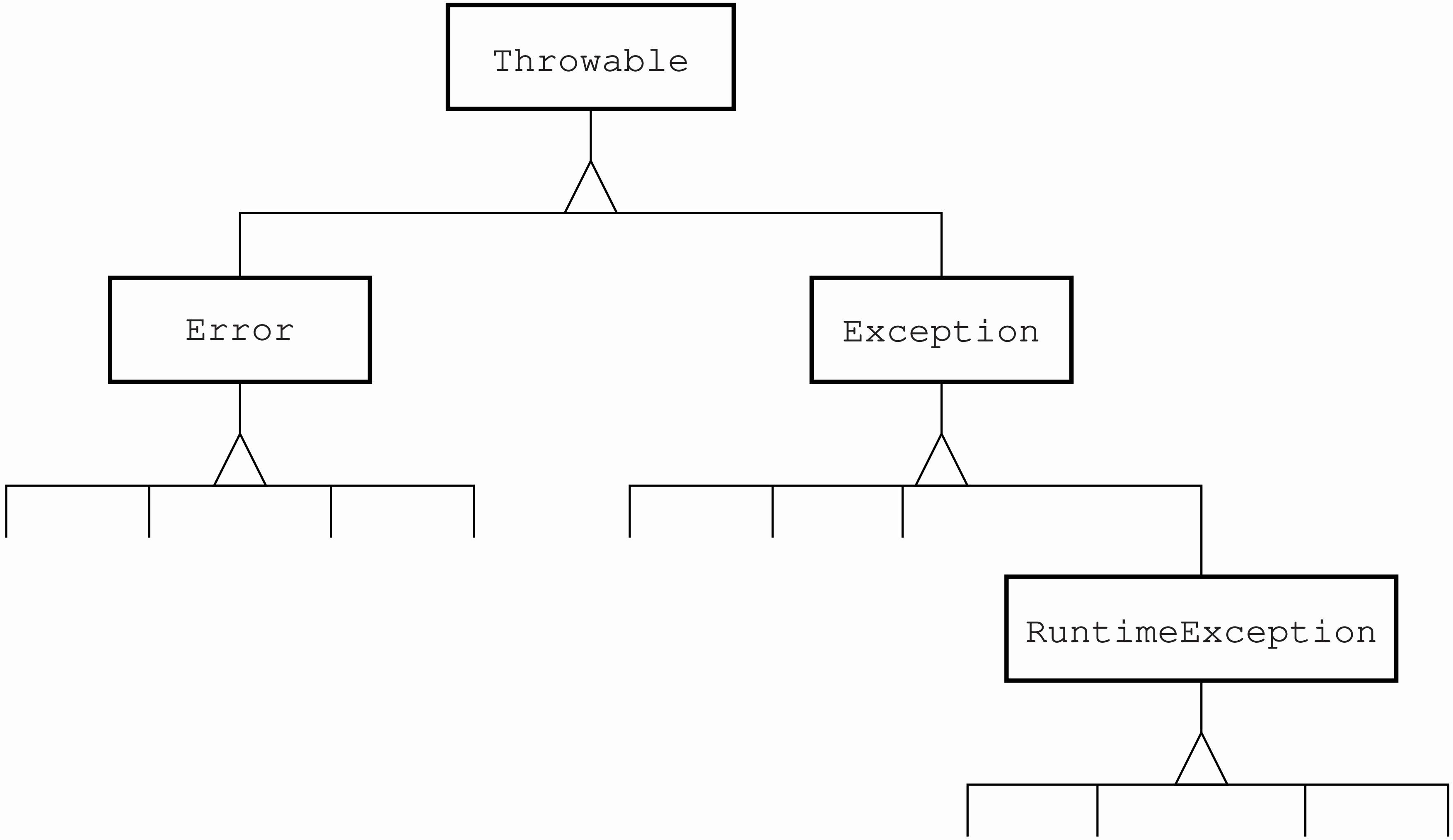
Note: When a method is overridden in a subclass, it must not throw exceptions not specified in the superclass.

There are two types of exceptions:

- **runtime exceptions** – exceptions that can occur almost everywhere, they are usually produced directly by the runtime system (arithmetic exceptions, pointer exceptions, indexing exceptions).
- **checked exceptions** – all other exceptions (including user defined exceptions).

The compiler checks that **checked exceptions** are either caught or specified. **Runtime exceptions** need not be caught or specified.

Hierarchy of Exceptions



Hierarchy of Exceptions (cont.)

Subclasses of **Throwable**:

- Subclasses of **Error** – exceptions of that indicates serious problems that a reasonable application should not try to catch.
- Subclasses of **Exception** – normal exceptions that a reasonable application might want to catch. User-defined exceptions should be subclasses of **Exception** (but not of **RuntimeException**).
- Subclasses of **RuntimeException** – runtime exceptions, usually produced by the runtime system. An application might want to catch them.

Note: The classes **Throwable**, **Error**, **Exceptions**, and **RuntimeException** are from the package **java.lang**.

It is not necessary to catch or specify subclasses of **Error** and **RuntimeException**. All other exceptions must be either caught or specified.

Hierarchy of Exceptions (cont.)

It is convenient to hierarchize exceptions using inheritance. This approach enables:

- grouping of error types
- error differentiation.

```
public class StackException extends Exception {
    public StackException(String message) {
        super(message);
    }
}
```

```
public class EmptyStackException extends StackException {
    public EmptyStackException() {
        super("The stack is empty.");
    }
}
```

Throwing an Exception

Any Java code can throw an exception using the **throw** statement:

```
throw someThrowableObject;
```

The **throw** statement requires a single argument - a throwable object.

An example of throwing an exception in an implementation of a stack:

```
public Object pop() throws EmptyStackException {
    if (size == 0) {
        throw new EmptyStackException();
    }
    Object obj = objectAt(size - 1);
    setObjectAt(size - 1, null);
    size--;
    return obj;
}
```

Class `Throwable`

The constructors of **java.lang.Throwable**:

- Throwable()
- Throwable(String message)
- Throwable(String message, Throwable cause)
- Throwable(Throwable cause)

The most important methods:

- String getMessage()
- Throwable getCause()
- Throwable initCause(Throwable cause)
- String toString()
- void printStackTrace()

Note: Every exception contains information about the call stack at the moment when the exception was created.

Class `Error`

An **Error** is a subclass of **Throwable** that indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions.

The most important subclasses (in the package **java.lang**):

- VirtualMachineError
  - OutOfMemoryError
  - StackOverflowError
  - InternalError
  - UnknownError
- LinkageError
- ThreadDeath
- AssertionError

Class `RuntimeException`

The most important subclasses of **java.lang.RuntimeException**:

- ArithmeticException
- IndexOutOfBoundsException
  - ArrayIndexOutOfBoundsException
  - StringIndexOutOfBoundsException
- IllegalArgumentException
  - NumberFormatException
- NullPointerException
- ClassCastException
- NegativeArraySizeException
- ArrayStoreException
- IllegalStateException
- UnsupportedOperationException

Exception Advantages

The use of exceptions has the following advantages over traditional error management techniques:

- **Separating error handling code from regular code**
  A problem which can raise at many places in program can be handled in only one place.
- **Propagating errors up the call stack**
  Mechanism enabling propagation of exceptions over the call stack enables transparent handling of errors raised in libraries.
- **Grouping error types and error differentiation**
  Multiple types of errors can be handled similarly at one place.