

Exception

”Exception is a clean way to check for errors without clustering code.” (Arnold and Gosling)

Not really exceptional – SPEC JVM 98, javac: 23,373 throws; jack: 241,877 throws.

Program abnormalities are objects of the `java.lang.Throwable` class; divided into two groups:

1. `java.lang.Error`: non-recoverable
2. `java.lang.Exception`: further divided as
 - `RuntimeException`
 - other exceptions

`RuntimeExceptions` are those that occur in the JVM, such as division-by-zero, null pointers, array index out of bound, etc. Since these exceptions are too numerous, Java provides default handlers for them and a programmer may ignore them in the program.

Other exceptions are less common and the programmer must clearly specify how to handle them (otherwise the compiler simply refuses to compile the program). Hence they are called *checked exceptions*.

Excetion handlers and the throws clause

A method must provide a handler for every checked exception that may occur during its execution or it must specify that it may throw the checked exception with the `throws` clause.

Exceptions are passed along the calling chain, until a handler for it is found. There is a handle for all kinds of exceptions in the JVM, which will catch all exceptions not handled by the application programs.

```
FileInputStream fis; // work018
```

```
void openFile(String name) throws FileNotFoundException {  
    // must declare the throws clause because the exception  
    // is a checked one and is not caught in openFile.  
    fis = new FileInputStream(name);  
}
```

```
void openDefault() {  
    try {  
        fis = new FileInputStream("default.file");  
    } catch (FileNotFoundException f) {  
        System.err.println("Cannot open default.file.");  
        System.exit(-1);  
    }  
}
```

```
void openOrDefault() {  
    try {  
        openFile("optional.file");  
    } catch (FileNotFoundException e) {  
        openDefault();  
    }  
}
```

```
void openOrHalt() {  
    try {  
        openFile("required.file");  
    } catch (FileNotFoundException e) {  
        System.err.println("required.file does not exist.");  
        System.exit(-1);  
    }  
}
```

Nested handlers

Inside a handler, we could also define other handlers.

```
// work043, chap 9
// all checked exceptions must be caught or thrown
import java.applet.Applet; import java.io.*;

public class CheckedException {
    FileInputStream fis;

    void openFile1() {
        try {
            fis = new FileInputStream("default.file");
        } catch (FileNotFoundException f) {
            // This catch FileNotFoundException must be included
            // otherwise the following error occurs:
            // Error: c:\lazywork\work043\CheckedException.java(14):
```

```

    // Exception java.io.FileNotFoundException must be caught,
    // or it must be declared in the throws clause of this method.
    System.err.println("Cannot open default.file.  Program stops");
} finally {
    System.err.println("in finally block");
}
}

```

```

void openFile2() {
    try {
        fis = new FileInputStream("default.file");
    } catch (FileNotFoundException e) {
        try {
            fis = new FileInputStream("optional.file");
        } catch (FileNotFoundException f) { // must use f rather than e
            // This catch FileNotFoundException must ALSO be included

```

```
    // otherwise the following error occurs:
    // Error: c:\lazywork\work043\CheckedException.java(14):
    // Exception java.io.FileNotFoundException must be caught,
    // or it must be declared in the throws clause of this method
    System.err.println("in finally block");
}
} finally {
    System.err.println("in finally block");
}
}
public static void main(String[] args) {
    openFile1();
    openFile2();
}
} // end of class CheckedException
```

Inheritance hierarchy of Exceptions

Exceptions are also objects of a certain class. Here is the inheritance hierarchy. A programmer may define his own exception by subclassing a suitable exception class.

1. Object

- Throwable

- (a) Error

- LinkageError
- ThreadDeath
- VirtualMachineError
- AWTError

- (b) Exception

- ClassNotFoundException
- CloneNotSupportedException
- IllegalAccessException

- InstantiationException
- InterruptedException
- NoSuchMethodException
- RuntimeException
 - * EmptyStackException
 - * NoSuchElementException
 - * ArithmeticException
 - * ArrayStoreException
 - * ClassCastException
 - * IllegalArgumentException
 - i. IllegalThreadStateException
 - ii. NumberFormatException
 - * IllegalMonitorStateException
 - * IndexOutOfBoundsException
 - i. ArrayIndexOutOfBoundsException
 - ii. StringIndexOutOfBoundsException

- * NegativeArraySizeException
- * NullPointerException
- * SecurityException
- AWTException
- IOException
 - * EOFException
 - * FileNotFoundException
 - * InterruptedIOException
 - * UTFDataFormatException
 - * MalformedURLException
 - * ProtocolException
 - * SocketException
 - * UnknownHostException
 - * UnknownServiceException

Order of handlers

According to the semantics of inheritance, every `FileNotFoundException` is also an `IOException`, which in turn is also an `Exception`.

Therefore a handler for `IOException` could also handle a `FileNotFoundException`.

When an exception occurs, the JVM searches the list of all handlers from top to bottom in the method. The first matching handler will be employed. Thus the order of handlers does matter.

In the following code, the second handler (for `FileNotFoundException`) is useless since every `FileNotFoundException` will be caught by the first handler (for `IOException`).

```
try {  
    FileInputStream pF = new FileInputStream("message.txt");
```

```
Properties prop = new Properties();
prop.load(pF);
prop.list(new PrintStream(
    new FileOutputStream("prop.out")));
} catch (java.io.IOException e) {
    System.err.println("IO failed");
} catch (java.io.FileNotFoundException e) {
    System.err.println("file not found");
}
```

A programmer-defined exception

A programmer-defined exception is in the next page.

Programmer-defined exceptions are especially useful in third-party packages. Designers should consider all possible exceptions that may occur in their packages, the proper handlers for them, and, if necessary, throw the exceptions back to routines using the packages.

The finally block

Exceptions are *exceptional*. Their occurrences immediately interrupt the `try` block. Hence, we sometimes need to clean up some partially done work after an exception occurs. We can do it with the `finally` block.

A `finally` block is attached to the last exception handler.

In the following example, the `writelists` may complete in three ways:

1. `pStr` is opened and the 9th element is printed successfully. After the `try` block finishes, the `finally` block will be executed, during which `pStr` is closed.
2. An `IOException` occurs. The `pStr.println` command comes to a halt immediately. Then the `IOException` handler is executed. The `finally` block will be executed after the handler is done.

3. An `ArrayIndexOutOfBoundsException` occurs. The offending `vector.elementAt(9)` instruction halts immediately. Since there is no appropriate handler, the `finally` block is executed and then the exception is passed back to the caller of `writelist`.

```
public void writelist() throws ArrayIndexOutOfBoundsException {
    PrintStream pStr = null;
    try {
        pStr = new PrintStream(
            new BufferedOutputStream(
                new FileOutputStream("outfile")));
        pStr.println("The 9th element is " +
            vector.elementAt(9));
        // do not need      pStr.close()      here
        // since close() is done at the finally-block
    } catch (IOException e) {
```

```
        System.err.println("i/o error");  
    } finally {  
        if (pStr != null) pStr.close();  
    }  
}
```

Here is a more complete example of programmer-defined exceptions and the finally block.

```
import java.applet.Applet; // work054
import java.awt.Graphics; import java.io.*; import java.util.*;

class PrivateException extends Exception {
    public PrivateException() { };
    public String getMessage(){return "a private exception";}
}

public class TestFinalize extends Applet{
    void noException() {
        try {
            int i = 1;
            if (i == 1) { i ++;
            } else { throw (new PrivateException()); }
        }
    }
}
```

```
} catch (PrivateException e) {
    System.err.println("Caught an exception:"+e.getMessage());
} finally {
    System.err.println("finally block in noException method.");
}
}
```

```
void throwException() throws PrivateException {
    try {
        throw (new PrivateException());
    } finally {
        System.err.println("finally in throwException method.");
    }
}
```

```
void caughtException() {
```

```
try {
    throw (new PrivateException());
} catch (PrivateException e) {
    System.err.println("Caught an exception:"+e.getMessage());
} finally {
    System.err.println("finally in caughtException method.");
}
}

public void paint(Graphics g) {
    noException();
    try {
        throwException();
    } catch (PrivateException e) {
        System.err.println("Exception in paint():"+e.getMessage());
    } finally {
```

```
        System.err.println("finally after calling throwException()");
    }
    caughtException();
}
} // end of TestFinalize
```

```
// output is as follows:
// finally block in noException method.
// finally block in throwException method.
// Caught an exception in paint():a private exception
// finally block after call to throwException()
// Caught an exception:a private exception
// finally block in caughtException method.
// end of output
```