Unit objectives

- After completing this unit, you should be able to:
  - Explain what a Java exception is and describe the benefits of object-oriented exception handling
  - Describe the conditions that act as the source of exceptions
  - Use try/catch/finally blocks to catch and handle specific exceptions
  - Use the throw keyword to throw a predefined Throwable object or your own Exception subtype

Exceptions

- An *exception* is an event or condition that disrupts the normal flow of execution in a program
  - Exceptions are errors in a Java program
  - The condition causes the system to *throw* an exception
  - The flow of control is interrupted and a handler will *catch* the exception

Some drawbacks in older programming languages

- Example(following codes are to read a file into memory)

Are these codes robust?

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

Some drawbacks in older programming languages

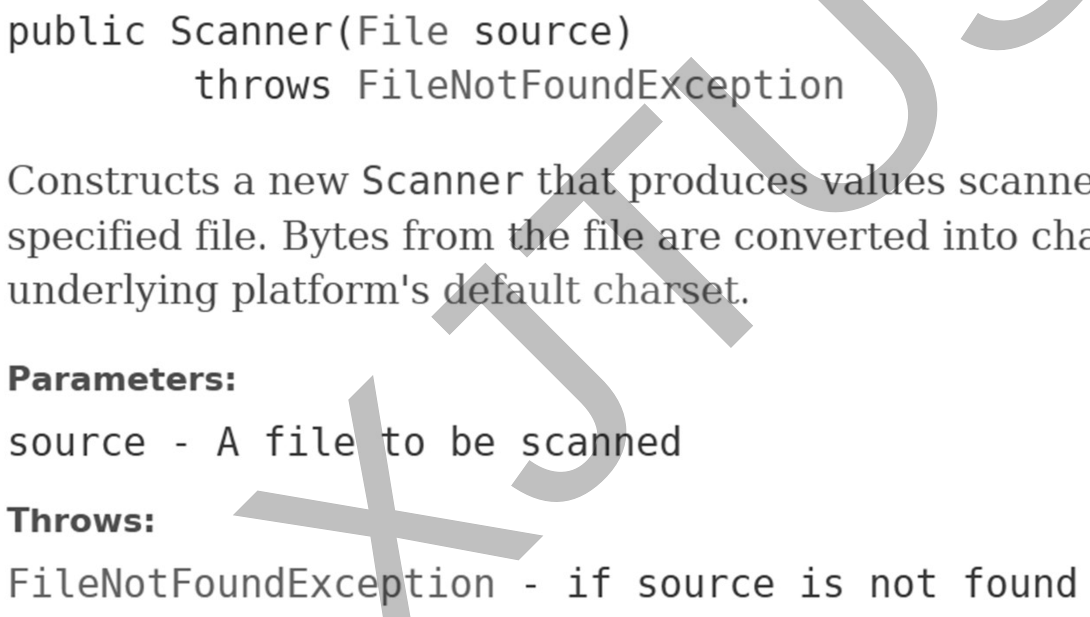
```
errorCodeType readFile( ) {
     initialize errorCode = 0;
     open the file;
     if (theFileIsOpen) {
          determine the length of the file;
          if (gotTheFileLength) {
                 allocate that much memory;
                 if (gotEnoughMemory) {
                       read the file into memory;
                            if (readFailed) { errorCode = -1; }
                } else { errorCode = -2; }
         } else { errorCode = -3; }
         close the file;
         if (theFileDidntClose && errorCode == 0) { errorCode = -4; }
         else { errorCode = errorCode and -4; }
      } else { errorCode = -5; }
      return errorCode; }
```

Java's advantages in excpetion handling

- You will be informed of the exceptional conditions that may arise in calling a method
  - Exceptions are declared in the method's signature
- You are forced to handle exceptions while writing the main logic and cannot leave them as an afterthought
  - Your program cannot compiled without the exception handling codes
- Exception handling codes are separated from the main logic
  - Via the try-catch-finally construct

Demo

- Example: java.util.Scanner
  - The signature of the Scanner’s constructor with a File argument is given



```
public Scanner(File source)
        throws FileNotFoundException
```

Constructs a new `Scanner` that produces values scanned from the specified file. Bytes from the file are converted into characters using the underlying platform's default charset.

Parameters:

`source - A file to be scanned`

Throws:

`FileNotFoundException - if source is not found`

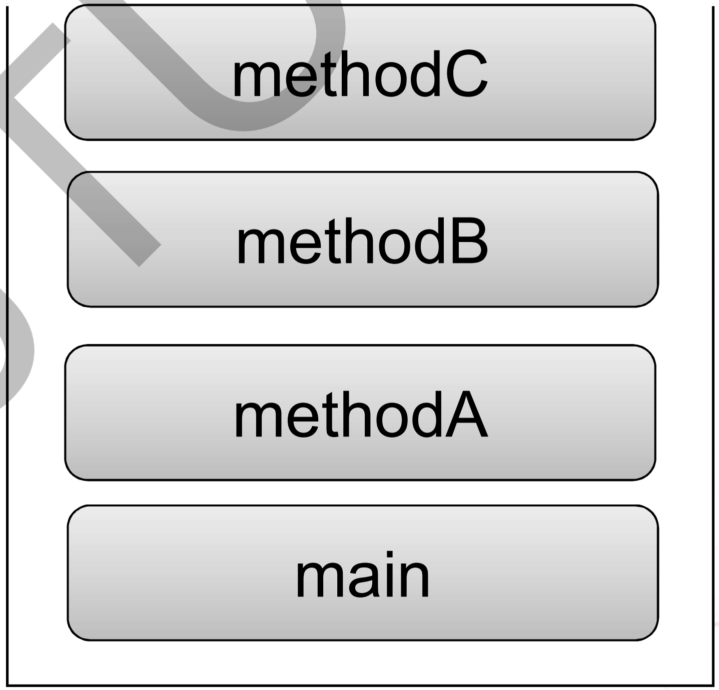
Seperating Error Handling Code from “Regular ” Code

- If we use java exception:

```
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; }
 }
```

Method Call Stack

- A typical application involves many levels of method calls, which is managed by a so-called method call stack
  - A stack is a last-in-first-out queue
- To run Demo



Propagating Errors Up the Call Stack

- If readFile() method is called by the following code

```
method1( ) {
    method2( ); }
method2( ) {
    method3( ); }
method3( ){
    readFile( ); }
```

- When errors of readFile() happen,only method1() want to know what error code of readFile() is.

Propagating Errors Up the Call Stack

```
method1( ) {
        errorCodeType error;
        error = method2( );
        if (error) doErrorProcessing
        else proceed; }
errorCodeType method2( ) {
        errorCodeType error;
        error = method3( );
        if (error)  return error;
        else proceed; }
errorCodeType method3 ( ){
        errorCodeType error;
        error = readFile( );
        if (error) return error;
        else proceed; }
```

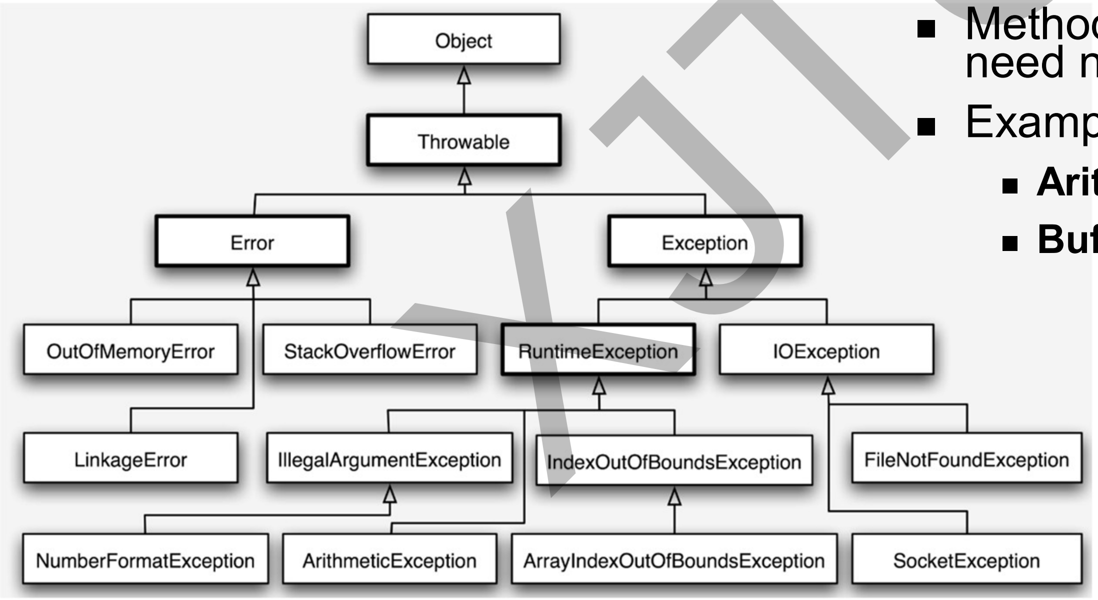
```
method1( ) {
   try {
          method2( );
        }
   catch (exception) {
        doErrorProcessing;
   }
 }
method2( ) throws exception {
     method3( ); }
method3( ) throws exception {
    readFile( ); }
```

Exception handling

- Exception handling is object-oriented
  - It encapsulates unexpected conditions in an object
  - It provides an elegant way to make programs robust
  - It isolates abnormal from regular flow of control

The exception hierarchy

- **Throwable** is the base class, and provides a common interface and implementation for most exceptions
- **Error** indicates serious problems that a reasonable application should not try to catch, such as:
  - **VirtualMachineError**
  - **CoderMalfunctionError**
- **Exception** heads the class of conditions that should usually be either caught or specified as thrown
- A **RuntimeException** can be thrown during the normal operation of the JVM
  - Methods may choose to catch these but need not specify them as thrown
  - Examples:
    - **ArithmeticException**
    - **BufferOverflowException**

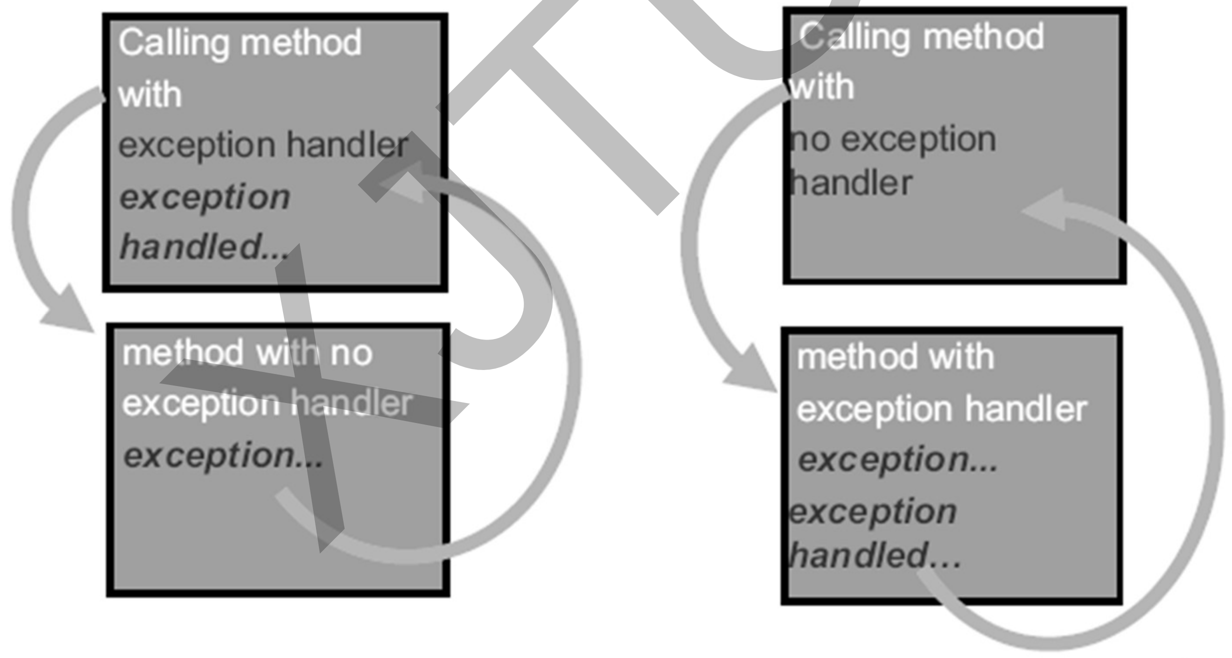


The Throwable Class

```
public class java.lang.Throwable extends Object  implements java.io.Serializable {
    public Throwable();
    public Throwable(String msg);
    public Throwable(String msg, Throwable cause);
    public Throwable(Throwable cause);
    public String getMessage();
    public String getLocalizedMessage();
    public Throwable getCause();
    public Throwable initCause(Throwable cause);
    public String toString();
    public void printStackTrace();
    public void printStackTrace(java.io.PrintStream);
    public void printStackTrace(java.io.PrintWriter);
    public Throwable fillInStackTrace();
    public StackTraceElement[] getStackTrace();
    public void setStackTrace(StackTraceElement[] stackTrace);
}
```

Handling exceptions

- Checked exceptions must be either handled in the method where they are generated, or delegated to the calling method

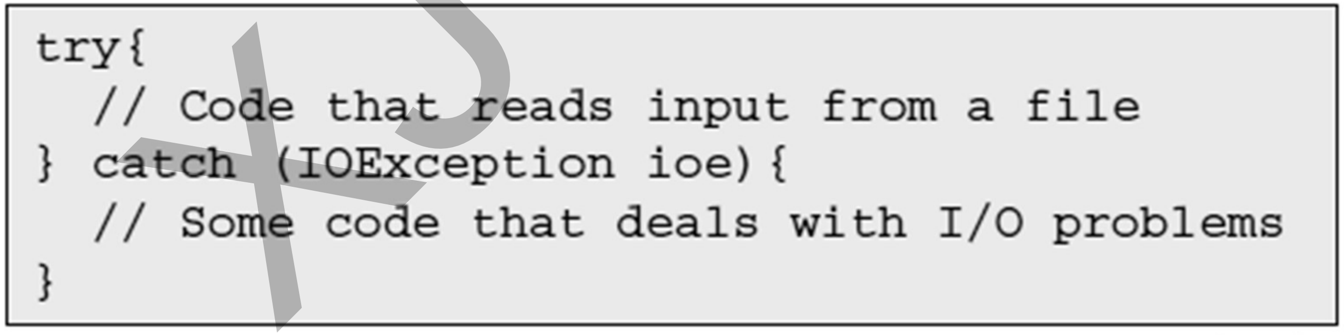


Keywords

- **throws**
  - A clause in a method declaration that lists exceptions that may be delegated up the call stack
    - Example: public int doIt() throws SomeException, …
- **try**
  - Precedes a block of code with attached exception handlers
  - Exceptions in the try block are handled by the exception Handlers
- **catch**
  - A block of code to handle a specific exception
- **finally**
  - An optional block which follows catch clauses
  - Always executed regardless of whether an exception occurs
- **throw**
  - Launches the exception mechanism explicitly
    - Example: throw (SomeException)

try/catch blocks

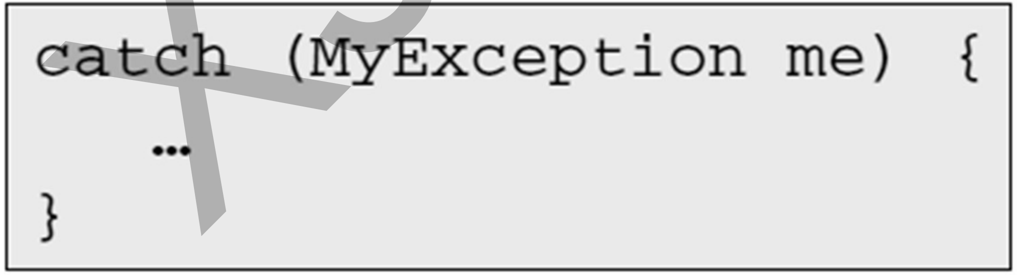
- To program exception handling, you must use try/catch blocks
- Code that might produce a given error is enclosed in a try block
- The catch clause must immediately follow the try block



```
try{
  // Code that reads input from a file
} catch (IOException ioe){
  // Some code that deals with I/O problems
}
```

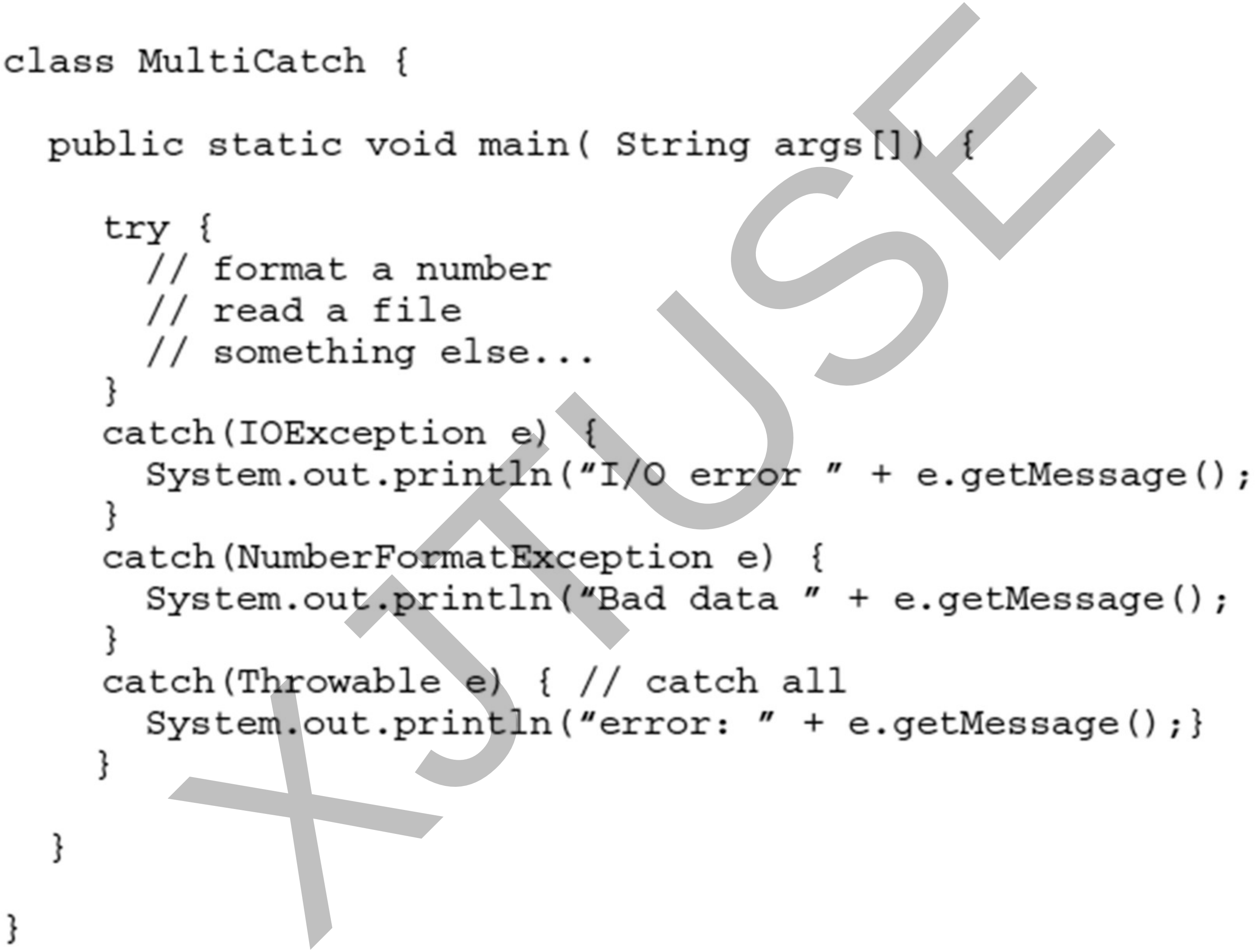
The catch clause

- The clause always has one argument that declares the type of exception to be caught
- The argument must be an object reference for the class **Throwable** or one of its subclasses
- Several catch clauses may follow one try block



```
catch (MyException me) {
    ...
}
```

Example



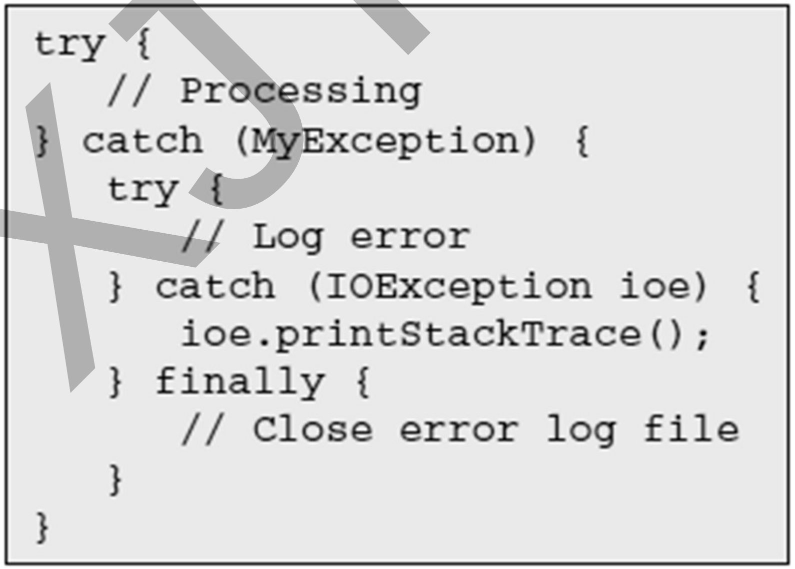
```
class MultiCatch {

  public static void main( String args[]) {

     try {
       // format a number
       // read a file
       // something else...
     }
     catch(IOException e) {
       System.out.println(”I/O error ” + e.getMessage();
     }
     catch(NumberFormatException e) {
       System.out.println(”Bad data ” + e.getMessage();
     }
     catch(Throwable e) { // catch all
       System.out.println(”error: ” + e.getMessage();}
     }

  }

}
```

The finally clause

- Optional clause that allows cleanup and other operations to occur whether an exception occurs or not
  - May have try/finally with no catch clauses
- Executed after any of the following:
  - try block completes normally
  - catch clause executes
    - Even if catch clause includes return
  - Unhandled exception is thrown, but before execution returns to calling method

Nested exception handling

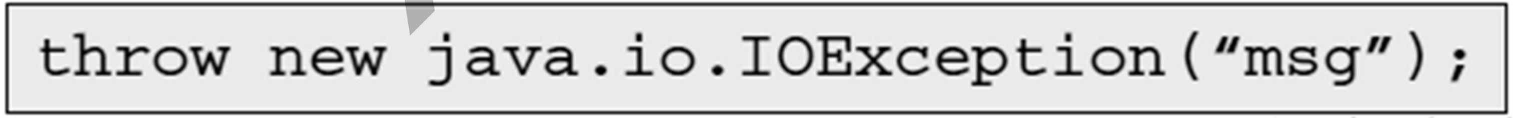
- It may be necessary to handle exceptions inside a catch or finally clause
  - For example, you may want to log errors to a file, but all I/O operations require **IOException** to be caught.
- Do this by nesting a try/catch (and optional finally) sequence inside your handler



```
try {
   // Processing
} catch (MyException) {
   try {
      // Log error
   } catch (IOException ioe) {
      ioe.printStackTrace();
   } finally {
      // Close error log file
   }
}
```

The throw keyword

- Not to be confused with keyword throws
- Can be used in a try block when you want to deliberately throw an exception
- You can throw a predefined Throwable object or your own Exception subtype
- Create a new instance of the exception class to encapsulate the condition
- The flow of the execution stops immediately after the throw statement, and the next statement is not reached
  - A finally clause will still be executed if present



```
throw new java.io.IOException(“msg”);
```

Unit summary

- In this unit, you should have learned to:
  - Explain what a Java exception is and describe the benefits of object-oriented exception handling
  - Describe the conditions that act as the source of exceptions
  - Use try/catch/finally blocks to catch and handle specific exceptions
  - Use the throw keyword to throw a predefined Throwable object or your own Exception subtype
  - Describe and use assertions