Exception handling

Exception-Handling Fundamentals

- An **exception** is an abnormal condition that arises in a code sequence at run time.
- In other words, an exception is a **run-time error**.

- A **Java exception is an object** that describes an exceptional (i.e. error) condition that has occurred in a piece of code.
- When an **exceptional condition arises**, an **object** representing that exception is **created and thrown in the method** that caused the error.

- That **method may choose to handle the exception itself**, or pass it on.
- Either way, at some point, **the exception is caught and processed**.

- Exceptions can be **generated by the Java run-time system**, or they **can be manually generated by your code**.
  - **Exceptions thrown by Java** relate to **fundamental errors** that **violate the rules of the Java language** or the constraints of the Java execution environment.
  - **Manually generated** exceptions are typically **used to report some error condition** to the caller of a method.
- Java exception handling is managed via five keywords: **try, catch, throw, throws,** and **finally.**

- **Try**: Contains program statements that you want to monitor for exceptions.
- **Catch:** Catch the exception and handle it in some rationale manner.
- **Throw:** Manually throws an exception.
- **Throws:** To specify any exception that is thrown out of a method
- **Finally:** Contains any code that absolutely must be executed before a method returns. Guaranteed to run whether or not an exception occurs.

General form

```
try {
    // block of code to monitor for errors
    }
catch (ExceptionType1 exOb) {
    // exception handler for ExceptionType1
    }
catch (ExceptionType2 exOb) {
    // exception handler for ExceptionType2
    }
    // ...
finally {
    // block of code to be executed before try block ends
    }
```

Exception Types

- **All** exception types **are subclasses** of the built-in class **Throwable.**
- Thus, Throwable is at the top of the exception class hierarchy.

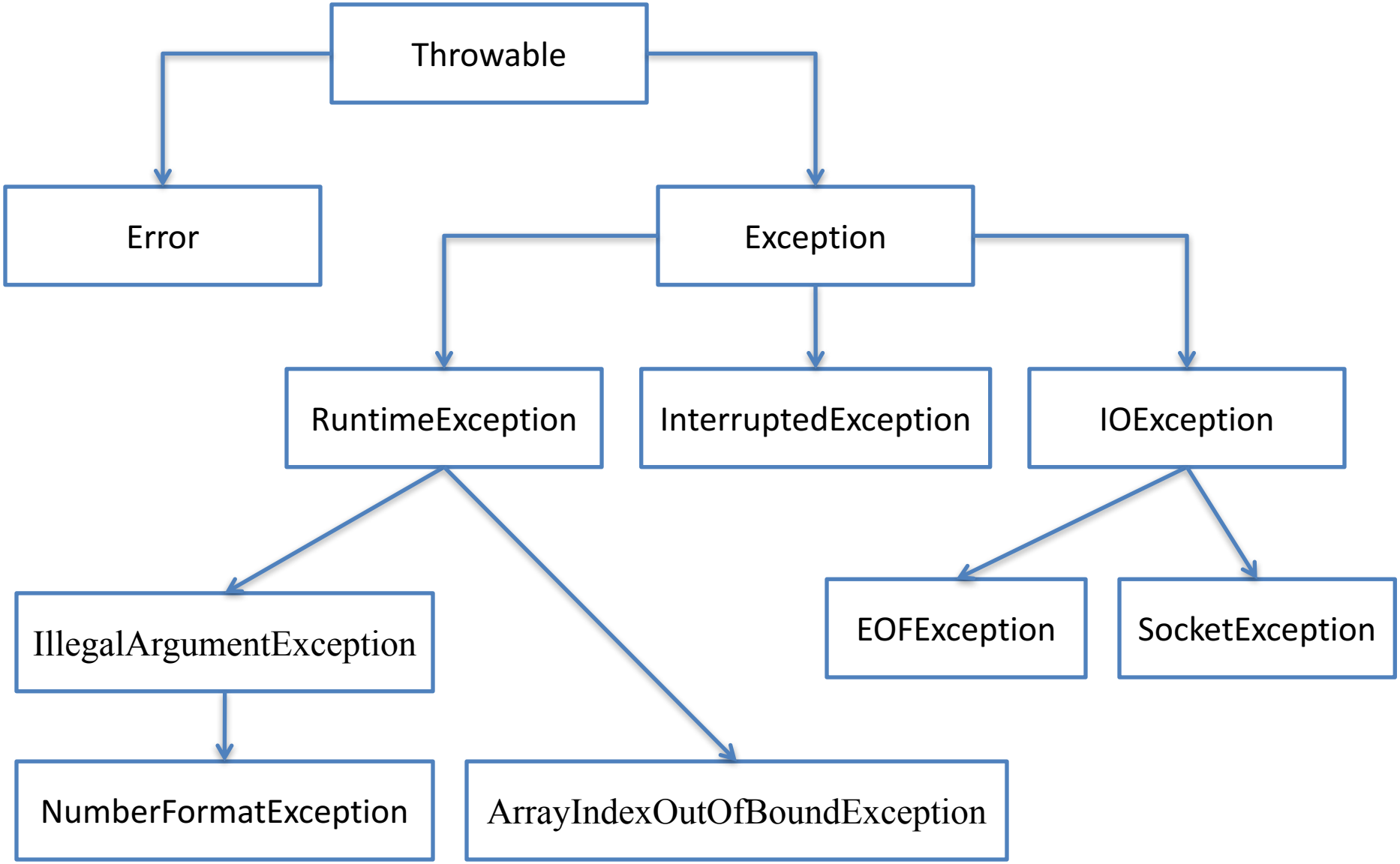
- **Exception** class is used for exceptional conditions that user programs should catch.
- This is also the class that you will subclass to create your own custom exception types.

- **Runtime Exceptions are automatically defined** for the programs that you write and include things such as division by zero and invalid array indexing.

- ***Error*** defines exceptions that are not expected to be caught under normal circumstances by your program.

- Exceptions of type ***Error*** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself.

- Stack overflow is an example of such an error.


Throwable
Error
Exception
RuntimeException
InterruptedException
IOException
IllegalArgumentException
EOFException
SocketException
NumberFormatException
ArrayIndexOutOfBoundException

Using try and catch

- **A try and its catch statement form a unit**.
- The **scope of the catch** clause is restricted to those statements specified by the immediately preceding try statement.
- A catch statement **cannot catch an exception thrown by another try** statement (**except** in the case of nested try statements)
- The statements that are protected by try must be surrounded by curly braces.
- You **cannot use try on a single statement**.

Displaying a Description of an Exception

- **Throwable** overrides the **toString( )** method so that it returns a string containing a description of the exception.

- You can display this description in a **println( )** statement by simply passing the exception as an argument.

Multiple catch Clauses

- In some cases, **more than one exception could be raised by a single piece of code.**
- To handle this type of situation, you can specify **two or more catch clauses**, each catching a different type of exception.

- When an exception is thrown, each catch statement is **inspected in order**, and the first one whose type matches that of the exception is executed.
- After one catch statement executes, the **others are bypassed**, and execution continues after the try/catch block.

- When you use multiple catch statements, it is important to remember that **exception subclasses must come before any of their superclasses.**

- This is because a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses.

- Thus, a subclass would never be reached if it came after its superclass.

- Further, in Java, **unreachable code is an error.**

Nested try Statements

- The try statement can be nested.
- That is, a try statement can be inside the block of another try.
- Each time a try statement is entered, the context of that exception is pushed on the stack.
- If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match.
- This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted.
- If no catch statement matches, then the Java run-time system will handle the exception.

- Nesting of try statements can occur in less obvious ways when method calls are involved.
- For example, you can enclose a call to a method within a try block.
- Inside that method is another try statement.
- In this case, the try within the method is still nested inside the outer try block, which calls the method.

throw

- It is possible for your program to throw an exception explicitly, using the **throw** statement.
- The general form of throw is shown here:

  throw *ThrowableInstance;*

- Here, *ThrowableInstance* must be an object of type Throwable or a subclass of Throwable.

- Simple types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions.
- There are two ways you can obtain a Throwable object: **using a parameter into a catch clause, or creating one with the new operator.**

- The flow of execution stops immediately after the throw statement; any subsequent statements are not executed.

- The nearest enclosing catch block is inspected to see if it has a catch statement that matches the type of the exception.

- If it does find a match, control is transferred to that statement.
- If not, then the next enclosing catch statement is inspected, and so on.
- If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

throws

- If a **method** is capable of **causing an exception that it does not handle**, it must specify this behavior so that callers of the method can guard themselves against that exception.
- You do this by including a **throws** clause in the method's declaration.
- A throws clause **lists the types of exceptions** that a method might throw.

- This is necessary for all exceptions, **except** those of type ***Error*** or ***RuntimeException***, or any of their subclasses.

- All other exceptions that a method can throw must be declared in the throws clause.
- If they are not, a compile-time error will result.

- This is the general form of a method declaration that includes a throws clause:

```
type method-name(parameter-list) throws exception-list
{
// body of method
}
```

- Here, exception-list is a comma-separated list of the exceptions that a method can throw.

finally

- When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method.
- Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely.
- This could be a problem in some methods.

- For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism.

- The **finally** keyword is designed to address this contingency.

- **finally** creates a **block of code** that will be **executed after a try/catch block has completed and before the code following the try/catch block.**

- The finally block will **execute whether or not an exception is thrown.**

- If an exception is thrown, the finally block will **execute even if no catch statement matches the exception.**

- Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also **executed just before the method returns.**

- This can be useful for **closing file handles** and **freeing up any other resources** that might have been allocated at the beginning of a method with the intent of disposing of them before returning.

- The finally clause is **optional**.

- However, **each try statement requires at least one catch or a finally clause.**

Creating Your Own Exception Subclasses

- You can also create your own exception types to handle situations specific to your applications.
- You need to define a **subclass of Exception** (which is, of course, a subclass of Throwable).
- Your subclasses don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions.

- The Exception **class does not define any methods of its own**.
- It does, of course, **inherit those methods** provided by Throwable.
- Thus, all exceptions, including those that you create, have the methods defined by Throwable available to them.
- They are shown in Table.

Method	Description
Throwable fillInStackTrace( )	Returns a **Throwable object that contains** a completed stack trace. This object can be rethrown
Throwable getCause( )	Returns the exception that underlies the current exception. If there is no underlying exception, **null is returned**
String getMessage( )	Returns a description of the exception.
void printStackTrace( )	Displays the stack trace.
void setStackTrace(StackTraceElement *elements[ ])*	Sets the stack trace to the elements passed in *elements. This method is for specialized* applications, not normal use.