Exception Handling

Siddharth Shah

Outline

- Concept: Exception and Exception Handling
- Exception Generation
- Five Key Words
- General Form of Exception Handling Block
- Exception Class Hierarchy and Exception Classes
- ***try*** and ***catch***
- Multiple catch Clauses
- Nested ***try*** Statements
- ***throw*** Statement

Outline

- ***throws*** Clause
- ***finally***
- Checked and Unchecked Exceptions
- Custom Exception
- Chained Exceptions
- ***multi-catch*** feature

Concept: Exception and Exception Handling

- A Java exception is an object that describes an exceptional (that is, 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.

Exception Generation

- 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

Five Key Words

- Java exception handling is managed via five keywords: **try, catch, throw, throws,** and **finally**.
- Program statements that you want to monitor for exceptions are contained within a try block. If an exception occurs within the **try** block, it is thrown.
- Your code can catch this exception (using **catch**) and handle it in some rational manner.

Five Key Words (Cont..)

- System-generated exceptions are automatically thrown by the Java runtime system. To manually throw an exception, use the keyword **throw**.
- Any exception that is thrown out of a method must be specified as such by a **throws** clause.
- Any code that absolutely must be executed after a try block completes, is put in a **finally** block.

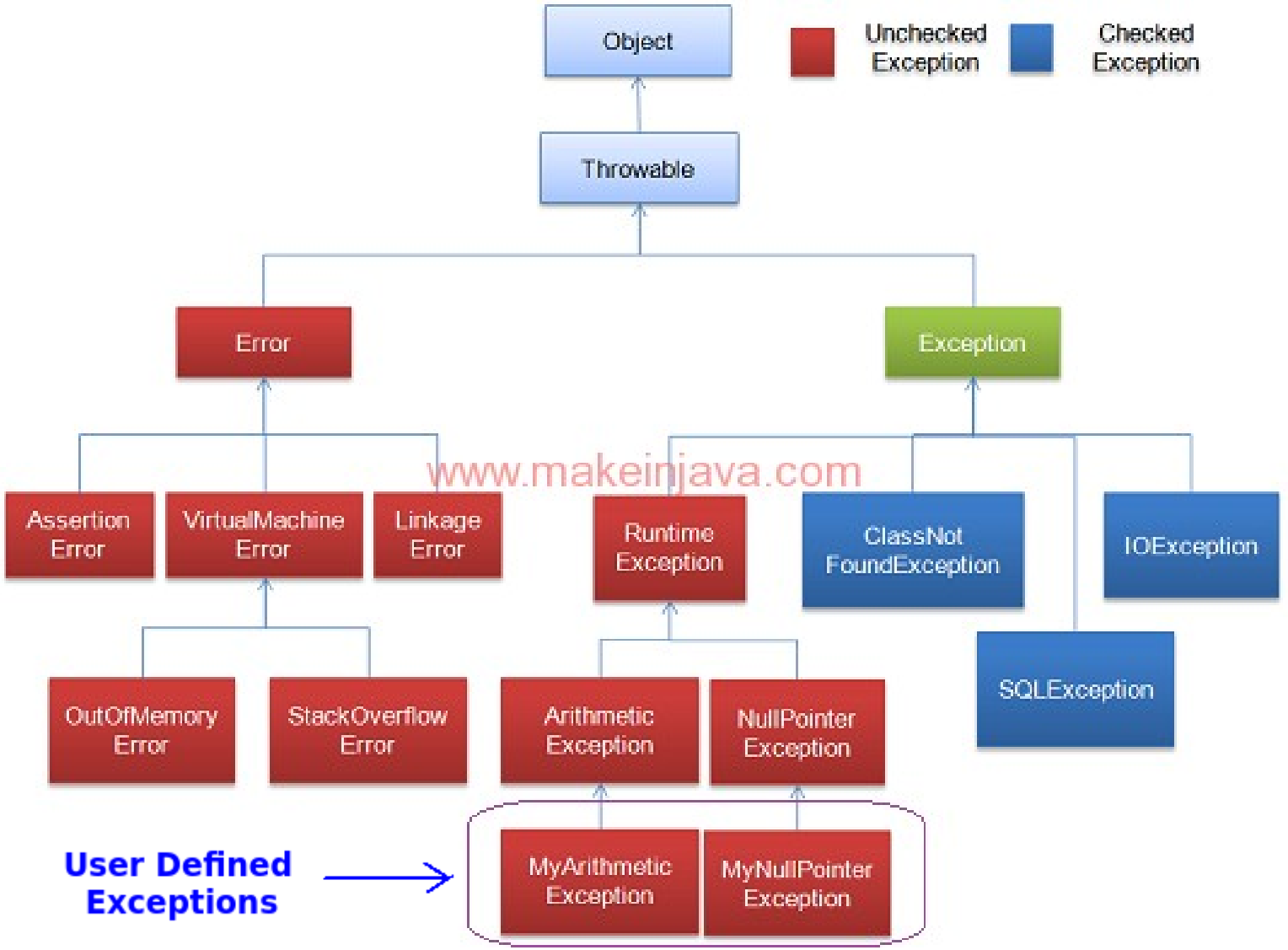
General Form of Exception Handling Block

```
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 after try block ends
}
```

Exceptions Class Hierarchy



Exception Classes

- **Throwable:** All exception types are subclasses of the built-in class Throwable
- **Exception:** This class is used for exceptional conditions that user programs should catch. This is also the class that can be subclassed to create custom exception types.
- **RuntimeException:** It is an important subclass of Exception. Exceptions of this type are automatically defined for the programs and include things such as division by zero and invalid array indexing.

Exception Classes (Cont..)

- **Error:** It defines exceptions that are not expected to be caught under normal circumstances by the 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.
- We will not be dealing with exceptions of type Error, because these are typically created in response to catastrophic failures that cannot usually be handled by the program.

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. (That is, they must be within a block.) You cannot use try on a single statement.
- The goal of most well-constructed catch clauses should be to resolve the exceptional condition and then continue on as if the error had never happened.

Multiple ***catch*** Clauses - 1

- 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.

Multiple ***catch*** Clauses - 2

- 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 - 1

- 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.

Nested *try* Statements - 2

- 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 Statement - 1

- 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;**
- ThrowableInstance must be an object of type Throwable or a subclass of Throwable.
- Primitive 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 in a catch clause
  - creating one with the new operator

throw Statement - 2

- The flow of execution stops immediately after the throw statement; any subsequent statements are not executed.
- The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception.
- If it does find a match, control is transferred to that statement. If not, then the next enclosing try block 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 Clause - 1

- 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

throws Clause - 2

- Below is 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 - 1

- **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.

finally - 2

- 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.

Checked and Unchecked Exceptions

- **Checked** are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword. e.g. **IOException**
- **Unchecked** are the exceptions that are not checked at compiled time, e.g. **NullPointerException**
- Exceptions under Error and RuntimeException classes are unchecked exceptions, everything else under Throwable is checked.

Creating Custom Exception - 1

- Although Java's built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications.
- This is quite easy to do: just 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.

Creating Custom Exception - 2

- 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.
- Exception defines four public constructors. Two support chained exceptions, described in the next section. The other two are shown here:
  - **Exception( ):** creates an exception that has no description.
  - **Exception(String msg):** lets you specify a description of the exception.

Chained Exceptions - 1

- The chained exception feature allows you to associate another exception with an exception. This second exception describes the cause of the first exception.

- For example, imagine a situation in which a method throws an **ArithmeticException** because of an attempt to divide by zero. However, the actual cause of the problem was that an **I/O error** occurred, which caused the divisor to be set improperly.

- Although the method must certainly throw an ArithmeticException, since that is the error that occurred, you might also want to let the calling code know that the underlying cause was an I/O error.

- Chained exceptions let you handle this, and any other situation in which layers of exceptions exist.

Chained Exceptions - 2

- To allow chained exceptions, two constructors and two methods were added to Throwable.
- The constructors are shown here:
  - **Throwable(Throwable causeExc):** causeExc is the exception that causes the current exception. That is, causeExc is the underlying reason that an exception occurred.
  - **Throwable(String msg, Throwable causeExc):** this form allows you to specify a description at the same time that you specify a cause exception.
- These two constructors have also been added to the **Error**, **Exception**, and **RuntimeException** classes.

Chained Exceptions - 3

- The chained exception methods supported by Throwable are getCause( ) and initCause( ).
  - **Throwable getCause( ):** returns the exception that underlies the current exception. If there is no underlying exception, null is returned.
  - **Throwable initCause(Throwable causeExc):** associates causeExc with the invoking exception and returns a reference to the exception. Thus, you can associate a cause with an exception after the exception has been created.
- However, the cause exception can be set only once. Thus, you can call initCause( ) only once for each exception object.
- Furthermore, if the cause exception was set by a constructor, then you can’t set it again using initCause( ).

multi-catch

- The **multi-catch** feature allows two or more exceptions to be caught by the same catch clause.
- It is not uncommon for two or more exception handlers to use the same code sequence even though they respond to different exceptions.
- Instead of having to catch each exception type individually, you can use a single catch clause to handle all of the exceptions without code duplication.
- To use a multi-catch, separate each exception type in the catch clause with the OR operator.
- Each multi-catch parameter is implicitly final, and hence it can’t be assigned a new value.

Reference

Book:

Java - The Complete Reference (Ninth Edition)