Introduction to JAVA Programming

Rathindra Nath Dutta

Senior Research Fellow
Advanced Computing & Microelectronics Unit
Indian Statistical Institute, Kolkata

October 15, 2020

Outline

1. Exception Handling
   - Basics
   - Exception Hierarchy
   - Uncaught Exceptions
   - `try` and `catch`
   - Nested `try` Statements
   - `throw` Statements
   - `throws` Clause
   - `finally` Clause
   - Checked and Unchecked Exceptions
   - Creating Own Exception Classes
   - Chained Exceptions

Exceptions

- An abnormal condition that arises in a code sequence at run time
- A run-time anomaly

- Traditionally these were dealt manually
- Typically some error codes were used

- Exceptions handling is OOP's way of run-time error management

- Common exceptions are: division by zero, using out of bounds indices of an array, accessing members of a null object etc.

Exceptions in Java

- Exception is an object, describing an exceptional/error condition

Exceptions in Java

- Exception is an object, describing an exceptional/error condition
- When an exceptional condition arises, an object representing that exception is created and *thrown* in the method

Exceptions in Java

- Exception is an object, describing an exceptional/error condition
- When an exceptional condition arises, an object representing that exception is created and *thrown* in the method
- That method may choose to handle the exception itself, or pass it on to its calling method
- Either way, at some point, the exception is *caught* and processed

Exceptions in Java

- Exception is an object, describing an exceptional/error condition
- When an exceptional condition arises, an object representing that exception is created and *thrown* in the method
- That method may choose to handle the exception itself, or pass it on to its calling method
- Either way, at some point, the exception is *caught* and processed
- Exceptions can be generated by the Java run-time system itself: denoting fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment
- They can be also manually generated to report some error condition to the caller of a method

Exception Handling in Java

- Java exception handling is managed via five keywords: `try`, `catch`, `throw`, `throws`, and `finally`

Exception Handling in Java

- 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

Exception Handling in Java

- 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 here, it is thrown automatically
- To handle this exception we catch and precess it in a `catch` block

Exception Handling in Java

- 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 here, it is thrown automatically
- To handle this exception we catch and precess it in a `catch` block
- To manually throw an exception, use the keyword `throw`

Exception Handling in Java

- 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 here, it is thrown automatically
- To handle this exception we catch and precess it in a `catch` block
- To manually throw an exception, use the keyword `throw`
- Any exception that is not handled locally, are thrown out of a method by a `throws` clause

Exception Handling in Java

- 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 here, it is thrown automatically
- To handle this exception we catch and precess it in a `catch` block
- To manually throw an exception, use the keyword `throw`
- Any exception that is not handled locally, are thrown out of a method by a `throws` clause
- Any code that absolutely must be executed after a try block completes is put in a `finally` block

Exception Types

- All exception types are subclasses of the built-in class `Throwable`

Exception Types

- All exception types are subclasses of the built-in class `Throwable`
- `Exception` is one subclass of `Throwable`

Exception Types

- All exception types are subclasses of the built-in class `Throwable`
- `Exception` is one subclass of `Throwable`
- `RuntimeException` is an important subclass of `Exception`
- Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero, invalid array indexing etc.

Exception Types

- All exception types are subclasses of the built-in class `Throwable`
- `Exception` is one subclass of `Throwable`
- `RuntimeException` is an important subclass of `Exception`
- Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero, invalid array indexing etc.
- Another subclass of `Throwable` is `Error`
- It defines exceptions that are not expected to be caught under normal circumstances by your program
- It is used by the Java run-time system to indicate errors having to do with the run-time environment, e.g. stack overflow etc.
- These are typically created in response to catastrophic failures that cannot usually be handled by our program

Uncaught Exceptions

- What happens when we don't handle exceptions and write the following:

```
class ABC {
    public static void main(String[] args) {
        int d = 0;
        int a = 42 / d;
    }
}
```

Uncaught Exceptions

- What happens when we don't handle exceptions and write the following:

```
class ABC {
    public static void main(String[] args) {
        int d = 0;
        int a = 42 / d;
    }
}
```

- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws it

Uncaught Exceptions

- What happens when we don't handle exceptions and write the following:

```
class ABC {
    public static void main(String[] args) {
        int d = 0;
        int a = 42 / d;
    }
}
```

- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws it
- This causes the execution of `ABC` to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately

Uncaught Exceptions

- But we haven't supplied any exception handlers here, so the exception is caught by the default handler provided by the Java run-time system
- Any exception that is not caught by your program will ultimately be processed by the *default handler*

Uncaught Exceptions

- But we haven't supplied any exception handlers here, so the exception is caught by the default handler provided by the Java run-time system
- Any exception that is not caught by your program will ultimately be processed by the *default handler*
- The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program

Uncaught Exceptions

- But we haven't supplied any exception handlers here, so the exception is caught by the default handler provided by the Java run-time system
- Any exception that is not caught by your program will ultimately be processed by the *default handler*
- The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program
- We get the following message:

```
java.lang.ArithmeticException: / by zero
        at ABC.main(ABC.java:4)
```

Uncaught Exceptions

```
class ABC {
    static void subroutine() {
        int d = 0;
        int a = 10 / d;
    }
    public static void main(String[] args) {
        ABC.subroutine();
    }
}
```

Uncaught Exceptions

```
class ABC {
    static void subroutine() {
        int d = 0;
        int a = 10 / d;
    }
    public static void main(String[] args) {
        ABC.subroutine();
    }
}
```

- Here we will get the following message:

```
java.lang.ArithmeticException: / by zero
        at ABC.subroutine(Exc1.java:4)
        at ABC.main(Exc1.java:7)
```

- This is known as *Stack Trace*, it helps debugging the code

Using `try` and `catch`

- The default exception handler provided by the Java run-time system is useful for debugging
- Handling exception ourselves not only fixes the error but also prevents the program from automatically terminating

Using `try` and `catch`

- The default exception handler provided by the Java run-time system is useful for debugging
- Handling exception ourselves not only fixes the error but also prevents the program from automatically terminating
- To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a `try` block
- Immediately following the `try` block, include a `catch` clause that specifies the exception type that you wish to catch

Using `try` and `catch`

- The default exception handler provided by the Java run-time system is useful for debugging
- Handling exception ourselves not only fixes the error but also prevents the program from automatically terminating
- To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a `try` block
- Immediately following the `try` block, include a `catch` clause that specifies the exception type that you wish to 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

Using `try` and `catch`

- The default exception handler provided by the Java run-time system is useful for debugging
- Handling exception ourselves not only fixes the error but also prevents the program from automatically terminating
- To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a `try` block
- Immediately following the `try` block, include a `catch` clause that specifies the exception type that you wish to 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
- 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

Using try and catch

```
class ABC {
    public static void main(String args[]) {
        try { // monitor a block of code.
            int d = 0;
            int a = 42 / d;
            System.out.println("This will not be printed.");
        } catch (ArithmeticException e) { // catch divide-by-z
            System.out.println("Division by zero");
        }finally{
            System.out.println("Finally");
        }
        System.out.println("After try/catch");
    }
}
```

- `finally` block is guaranteed to be executed even if an exception is raised and not caught: for more details refer here

Multiple `catch` Clauses

- More than one exception could be raised by a single piece of code
- We 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
- 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
- It is important to remember that exception subclasses must come before any of their superclasses

Multiple catch Clauses

```
class ABC {
    public static void main(String args[]) {
        try {
            int a = args.length;
            int b = 42 / a;
            int c[] = { 1 };
            c[42] = 99;
        } catch(ArithmeticException e) {
            System.out.println("Division by zero");
        } catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("Invalid array index");
        } catch(Exception e) { // order is important
            System.out.println("Other exception");
        }
        System.out.println("After try/catch");
    }
}
```

Nested `try` Statements

- A `try` statement can be inside the block of another
- Each time a `try` block 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 default handler will handle the exception
- Nesting of `try` statements can occur in less obvious ways when method calls are involved, ref: NestedException.java

`throw` Statements

- We were only catching exceptions that are thrown by the Java run-time
- It is possible for your program to throw an exception explicitly, using the `throw` statement

  ```
  throw throwableInstance;
  ```

- `throwableInstance` must be an object of type `Throwable` or a subclass of `Throwable`
- There are two ways you can obtain a `Throwable` object: using a parameter in a `catch` clause or creating one with the `new` operator
- Flow of execution stops immediately after the `throw` statement; any subsequent statements are not executed
- The nearest enclosing `try` block is inspected for a matching catch statement. If not found, then the next enclosing `try` statement is inspected, and so on. If no matching `catch` is found, the default exception handler halts the program and prints the stack trace

throw Statements

```
void foo() {
    try {
        throw new NullPointerException("demo");
    } catch(NullPointerException e) {
        System.out.println("Caught inside demoproc.");
        throw e; // rethrow the exception
    }
}
...
try {
    foo();
} catch(NullPointerException e) {
    System.out.println("Recaught" + e);
}
```

`throws` Clause

- 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
- It is done by including a `throws` clause in the method's declaration

  `throw throwableInstance;`
- 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
- Calling method must surround the call by appropriate try / catch statement or it must be declared with `throws`

```
void foo() throws IllegalAccessException {
    throw new IllegalAccessException("demo");
}
```

`finally` Clause

- When exceptions are thrown, execution in a method takes 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 and may lead to problems: e.g. unclosed files
- `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
- `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
- When 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` block is also executed just before the method returns

`finally` Clause

- It is 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
- for more details refer here and Finally.java

Built-in Exceptions

- The package `java.lang` provides several exception classes

Built-in Exceptions

- The package `java.lang` provides several exception classes
- The most general of these exceptions are subclasses of the standard type `RuntimeException`
- These exceptions need not be included in any method's `throws` list
- These are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions
- Unchecked exceptions defined in `java.lang` are listed in Table 1

Built-in Exceptions

- The package `java.lang` provides several exception classes
- The most general of these exceptions are subclasses of the standard type `RuntimeException`
- These exceptions need not be included in any method's `throws` list
- These are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions
- Unchecked exceptions defined in `java.lang` are listed in Table 1
- Table 2 lists those exceptions defined by `java.lang` that must be included in a method's throws list if that method can generate one of these exceptions and does not handle it itself
- These are called *checked exceptions*

Built-in Exceptions

- The package `java.lang` provides several exception classes
- The most general of these exceptions are subclasses of the standard type `RuntimeException`
- These exceptions need not be included in any method's `throws` list
- These are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions
- Unchecked exceptions defined in `java.lang` are listed in Table 1
- Table 2 lists those exceptions defined by `java.lang` that must be included in a method's throws list if that method can generate one of these exceptions and does not handle it itself
- These are called *checked exceptions*
- In addition to the exceptions in `java.lang`, Java defines several more that relate to its other standard packages

Unchecked Exceptions

Exception	Meaning
`ArithmeticException`	Arithmetic error, such as divide-by-zero
`ArrayIndexOutOfBoundsException`	Array index is out-of-bounds
`ArrayStoreException`	Assignment to an array element of an incompatible type
`ClassCastException`	Invalid cast
`EnumConstantNotPresentException`	An attempt is made to use an undefined enumeration value
`IllegalArgumentException`	Illegal argument used to invoke a method
`IllegalMonitorStateException`	Illegal monitor operation, such as waiting on an unlocked thread
`IllegalStateException`	Environment or application is in incorrect state
`IllegalThreadStateException`	Requested operation not compatible with current thread state
`IndexOutOfBoundsException`	Some type of index is out-of-bounds
`NegativeArraySizeException`	Array created with a negative size
`NullPointerException`	Invalid use of a null reference
`NumberFormatException`	Invalid conversion of a string to a numeric format
`SecurityException`	Attempt to violate security
`StringIndexOutOfBounds`	Attempt to index outside the bounds of a string
`TypeNotPresentException`	Type not found
`UnsupportedOperationException`	An unsupported operation was encountered

Table 1: Unchecked Exceptions defined in `java.lang`

Checked Exceptions

Exception	Meaning
`ClassNotFoundException`	Class not found
`CloneNotSupportedException`	Attempt to clone an object that does not implement the Cloneable interface
`IllegalAccessException`	Access to a class is denied
`InstantiationException`	Attempt to create an object of an abstract class or interface
`InterruptedException`	One thread has been interrupted by another thread
`NoSuchFieldException`	A requested field does not exist
`NoSuchMethodException`	A requested method does not exist
`ReflectiveOperationException`	Superclass of reflection-related exceptions

Table 2: Checked Exceptions defined in `java.lang`

Why not all exceptions are checked?

Unchecked Exceptions — The Controversy

When to choose checked and unchecked exceptions

Creating Exception Subclasses

- We might want to create your own exception types to handle situations specific to your applications
- define a subclass of `Exception`, which is, of course, a subclass of `Throwable`
- The 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
- Thus, all exceptions, including those that we create, have the methods defined by Throwable available to them
- We can, and sometimes should, override one or more of these methods to better suit our exception type, e.g. `toString()`
- The complete API documentation is available here: Throwable and Exception

Chained Exceptions

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

Chained Exceptions

- The chained exception feature allows you to associate another exception with an exception
- This second exception describes the cause of the first exceptional
- A method may throw an `ArithmeticException` because of an attempt to divide by zero
- However, the actual cause of the problem can be that an I/O error occurred, which caused the divisor to be set improperly
- Although the method must certainly throw an `ArithmeticException`, we might also want to let the calling code know that the underlying cause was an I/O error

- Chained exceptions let us systematically create this kind of layers of exceptions
- It was introduced in JDK 1.4

Chained Exceptions

- To allow chained exceptions, the following constructors and methods are provided by `Throwable` and `Exception`

```
Throwable(Throwable causeExc)
Throwable(String msg, Throwable causeExc)
Throwable initCause(Throwable causeExc)

Throwable getCause( )

Exception(String msg, Throwable causeExc)
protected Exception(String msg, Throwable causeExc, boolean er
Exception(Throwable causeExc)
```