Programming in Java

Lecture 14: Exception Handling

Mahesh Kumar
Assistant Professor (Adhoc)

Department of Computer Science
Acharya Narendra Dev College
University of Delhi

Course webpage
[http://www.mkbhandari.com/mkwiki]

Outline

1. Exception Handling Fundamentals
2. Exception Types
3. Java’s Built-in Exceptions
4. Creating Your Own Exceptions

Exception Handling

- The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained.

- 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.

- In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes, and so on.

- Java's exception handling avoids these problems and, in the process, brings run-time error management into the object-oriented world.

Exception Handling Fundamentals

- 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.
- 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.

Exception Handling Fundamentals

- Java exception handling is managed via five keywords:

1. ***try:*** Program statements(block of code) that you want to monitor for exceptions are contained within a **try** block.

2. ***catch:*** 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.

3. ***throw:*** System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword **throw**.

4. ***throws:*** Any exception that is thrown out of a method must be specified as such by a **throws** clause.

5. ***finally:*** Any code that absolutely must be executed after a try block completes is put in a **finally** block.

Exception Handling Fundamentals

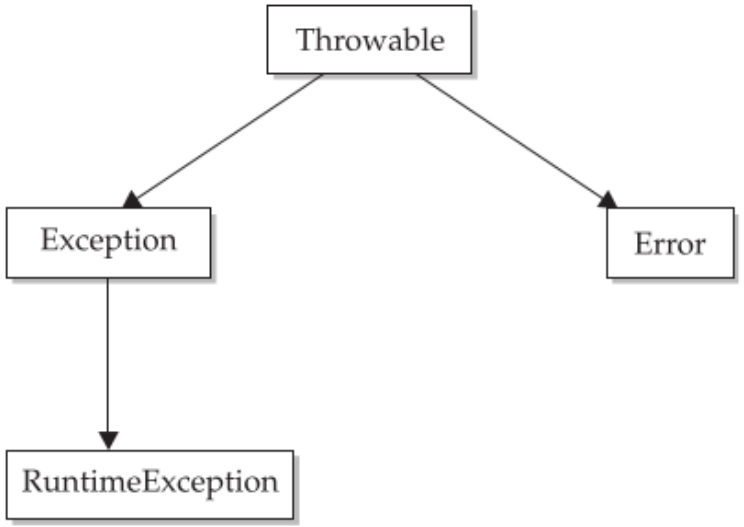
- This is the general form of an exception-handling block:

```
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
}
```

- Here, *ExceptionType* is the type of exception that has occurred.

Exception Types

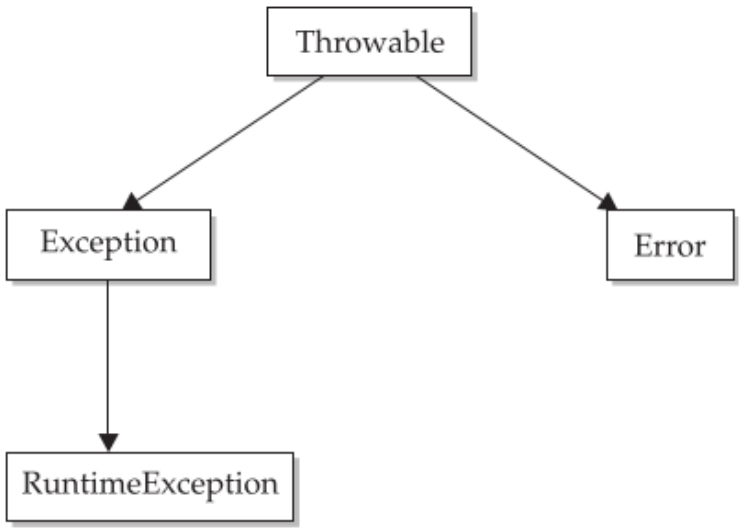
- The top-level exception hierarchy is shown here:



1. *All exception types are subclasses of the built-in class **Throwable** (is at the top of the exception class hierarchy).*
2. ***Throwable** partitions exceptions into two distinct branches using two subclasses, i.e. **Exception** and **Error.***
3. ***Exception Class***
   - *Used for exceptional conditions that user programs should catch.*
   - *To create your own custom exception types*
   - *There is an important subclass of Exception, called **RuntimeException**. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.*

Exception Types

- The top-level exception hierarchy is shown here:



4. ***Error***
   - *Which 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.*
   - *VirtualMachineError, OutOfMemoryError are examples of Error.*
5. ***Error vs. Exception***
   - *An Error indicates serious problem that a reasonable application should not try to catch.*
   - *Exception indicates conditions that a reasonable application might try to catch.*

Uncaught Exception

- What happens when you don't handle exceptions?
- This small program includes an expression that intentionally causes a divide-by-zero error:

```
class Exc0 {
        public static void main(String args[ ]) {
                Int d = 0;
                int a = 42 / d;
        }
}
```

1. When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception.
2. This causes the execution of Exc0 to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately.
3. In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system.

Uncaught Exception

- What happens when you don't handle exceptions?
- This small program includes an expression that intentionally causes a divide-by-zero error:

```
class Exc0 {
        public static void main(String args[ ]) {
                Int d = 0;
                int a = 42 / d;
        }
}
```

4. Any exception that is not caught by your program will ultimately be processed by the default handler.

5. 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.

6. Here is the exception generated when this example is executed:

```
java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:4)
```

Uncaught Exception

- What happens when you don't handle exceptions?
- This small program includes an expression that intentionally causes a divide-by-zero error:

```
class Exc0 {
        public static void main(String args[ ]) {
                Int d = 0;
                int a = 42 / d;
        }
}
```

- Here is the exception generated when this example is executed:

```
java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:4)
```

7 The simple ***stack trace*** for this program includes:

- **Class Name:** Exe0
- **Method Name:** main
- **File Name:** Exe0.java
- **Line Number:** 4
- **Type of Exception Thrown:** ArithmeticException

8 Java supplies several built-in exception types that match the various sorts of run-time errors that can be generated.

Uncaught Exception

- The *stack trace* will always show the sequence of method invocations that led up to the error.

```
class Exc1 {
        static void subroutine( ) {
                int d = 0;
                int a = 10 / d;
        }
        public static void main(String args[ ]) {
                Exc1.subroutine( );
        }
}
```

- The resulting stack trace from the default exception handler shows how the entire call stack is displayed:

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

1. The simple ***stack trace*** for this program includes:
   - **Class Name:** Exe1
   - **Method Name:** main, subroutine
   - **File Name:** Exe1.java
   - **Line Number:** 7, 4
   - **Type of Exception Thrown:** ArithmeticException

2. The call stack is quite useful for debugging, because it pinpoints the precise sequence of steps that led to the error.

Using try anc catch

- The default exception handler provided by the Java run-time system is useful for debugging.
- You will usually want to handle an exception yourself. Doing so provides two benefits:
  1. *It allows you to fix the error.*
  2. *It 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 anc catch

- The following program includes a ***try*** block and a ***catch*** clause that processes the ArithmeticException generated by the division-by-zero(DBZ) error:

```
class Exc2 {
public static void main(String args[ ]) {
int d, a;
try { // monitor a block of code.
        d = 0;
        a = 42 / d;
        System.out.println("This will not be printed.");
} catch (ArithmeticException e) { // catch DBZ error
        System.out.println("Division by zero.");
}

System.out.println("After catch statement.");
}
}
```

This program generates the following output:

```
Division by zero.
After catch statement.
```

1. Notice that the call to println( ) inside the ***try*** block is never executed.
2. Once an exception is thrown, program control transfers out of the ***try*** block into the ***catch*** block.
3. Once the ***catch*** statement has executed, program control continues with the next line in the program following the entire ***try/catch*** mechanism.

Using try anc 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.

Using try anc catch

```
// Handle an exception and move on.
import java.util.Random;
class HandleError {
        public static void main(String args[ ]) {
                int a=0, b=0, c=0;
                Random r = new Random( );
                for(int i=0; i<32000; i++) {
                        try {
                                b = r.nextInt( );
                                c = r.nextInt( );

                                a = 12345 / (b/c);

                        } catch (ArithmeticException e) {
                                System.out.println("Division by zero.");
                                a = 0; //Set a to zero and continue
                        }
                        System.out.println("a: " + a);
                }
        }
}
```

- If either division operation causes a divide-by-zero error, it is caught, the value of a is set to zero, and the program continues.
- **What will be the output ?**

Displaying a Description of an Exception

- ***Throwable*** overrides the toString( ) method (defined by ***Object***) 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 *(Displaying a description of an exception is valuable in experimenting with exceptions or debugging)*.

- For example, the ***catch*** block in the preceding program can be rewritten like this:

```
catch (ArithmeticException e) {
        System.out.println("Exception: " + e);
        a = 0; // set a to zero and continue
}
```

- When this version is substituted in the program, and the program is run, each divide-by-zero error displays the following message:

```
Exception: java.lang.ArithmeticException: / by zero
```

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.

Multiple *catch* Clauses

- The following example traps two different exception types:

```
// Demonstrate multiple catch statements.
class MultipleCatches {
        public static void main(String args[ ]) {
                try {
                        int a = args.length;
                        System.out.println("a = " + a);
                        int b = 42 / a;
                        int c[ ] = { 1 };
                        c[42] = 99;
                } catch(ArithmeticException e) {
                        System.out.println("Divide by 0: " + e);
                } catch(ArrayIndexOutOfBoundsException e) {
                        System.out.println("Array index oob: " + e);
                }
                System.out.println("After try/catch blocks.");
        }
}
```

- This program will cause a *division-by-zero exception* if it is started with no command-line arguments, since **a** will equal zero.

```
java MultipleCatches
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
```

Multiple *catch* Clauses

- The following example traps two different exception types:

```
// Demonstrate multiple catch statements.
class MultipleCatches {
        public static void main(String args[ ]) {
                try {
                        int a = args.length;
                        System.out.println("a = " + a);
                        int b = 42 / a;
                        int c[ ] = { 1 };
                        c[42] = 99;
                } catch(ArithmeticException e) {
                        System.out.println("Divide by 0: " + e);
                } catch(ArrayIndexOutOfBoundsException e) {
                        System.out.println("Array index oob: " + e);
                }
                System.out.println("After try/catch blocks.");
        }
}
```

- It will survive the division if you provide a command-line argument, setting **a** to something larger than zero.

```
java MultipleCatches  TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException:42
After try/catch blocks.
```

Multiple *catch* Clauses

- 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.

```
class SuperSubCatch {
public static void main(String args[ ]) {
try {
        int a = 0;  int b = 42 / a;
} catch(Exception e) {
        System.out.println("Generic Exception catch.");
} catch(ArithmeticException e) { // ERROR – unreachable
        System.out.println("This is never reached.");
}
}
}
```

1. Second ***catch*** statement is unreachable because the exception has already been caught.

2. Since ArithmeticException is a subclass of Exception, the first ***catch*** statement will handle all Exception-based errors, including ArithmeticException.

3. This means that the second ***catch*** statement will never execute. To fix the problem, reverse the order of the ***catch*** statements.

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.

Nested *try* Statements

```
// An example of nested try statements.
class NestTry {
        public static void main(String args[ ]) {
                try {
                        int a = args.length;
                        int b = 42 / a;
                        System.out.println("a = " + a);
                        try { // nested try block
                                if(a==1)
                                        a = a/(a-a); // division by zero
                                if(a==2) {
                                        int c[  ] = { 1 };
                                        c[42] = 99; // generate an out-of-bounds exception
                                }
                        } catch(ArrayIndexOutOfBoundsException e) {
                                System.out.println("Array index out-of-bounds: " + e);
                        }
                } catch(ArithmeticException e) {
                        System.out.println("Divide by 0: " + e);
                }
        }
}
```

- When you execute the program with no command-line arguments

```
java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero
```

Nested *try* Statements

```
// An example of nested try statements.
class NestTry {
        public static void main(String args[ ]) {
                try {
                        int a = args.length;
                        int b = 42 / a;
                        System.out.println("a = " + a);
                        try { // nested try block
                                if(a==1)
                                        a = a/(a-a); // division by zero*
                                if(a==2) {
                                        int c[  ] = { 1 };
                                        c[42] = 99; // generate an out-of-bounds exception
                                }
                        } catch(ArrayIndexOutOfBoundsException e) {
                                System.out.println("Array index out-of-bounds: " + e);
                        }
                } catch(ArithmeticException e) {
                        System.out.println("Divide by 0: " + e);
                }
        }
}
```

- When you execute the program with one command-line arguments

java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero

Since the inner block does not catch this exception, it is passed on to the outer ***try block, where it is handled.

Nested *try* Statements

```
// An example of nested try statements.
class NestTry {
        public static void main(String args[ ]) {
                try {
                        int a = args.length;
                        int b = 42 / a;
                        System.out.println("a = " + a);
                        try { // nested try block
                                if(a==1)
                                        a = a/(a-a); // division by zero
                                if(a==2) {
                                        int c[  ] = { 1 };
                                        c[42] = 99; // generate an out-of-bounds exception
                                }
                        } catch(ArrayIndexOutOfBoundsException e) {
                                System.out.println("Array index out-of-bounds: " + e);
                        }
                } catch(ArithmeticException e) {
                        System.out.println("Divide by 0: " + e);
                }
        }
}
```

- When you execute the program with two command-line arguments

java NestTry One Two

```
a = 2
Array index out-of-bounds:
        java.lang.ArrayIndexOutOfBoundsException:42
```

throw

- So far, you have only been catching exceptions that are thrown by the Java run-time system.
- However, 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.***
- 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:
  1. Using a parameter in a ***catch*** clause
  2. Creating one with the ***new*** operator.

throw

- 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 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.*

throw

```
/* Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler. */

class ThrowDemo {
        static void demoproc( ) {
                try {
                        //construct an instance of NullPointerException
                        throw new NullPointerException("demo");
                } catch(NullPointerException e) {
                        System.out.println("Caught inside demoproc.");
                        throw e; // rethrow the exception
                }
        }
        public static void main(String args[ ]) {
                try {
                        demoproc( );
                } catch(NullPointerException e) {
                        System.out.println("Recaught: " + e);
                }
        }
}
```

1. This program gets two chances to deal with the same error.
2. First, main( ) sets up an exception context and then calls demoproc( ).
3. The demoproc( ) method then sets up another exception-handling context and immediately throws a new instance of ***NullPointerException***, which is caught on the next line.
4. The exception is then rethrown.

This program generates the following output:

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

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.

throws

- 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.

throws

```
// Example of throws

class ThrowsDemo {

        static void throwOne( ) throws IllegalAccessException {
                System.out.println("Inside throwOne.");
                throw new IllegalAccessException("demo");
        }

        public static void main(String args[ ]) {
                try {
                        throwOne( );
                } catch (IllegalAccessException e) {
                        System.out.println("Caught " + e);
                }
        }
}
```

Here is the output of the program:

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

throw vs. throws

throw	throws
1. Java throw keyword is used to explicitly throw an exception	1. Java throws keyword is used to declare an exception.

2. void m(){
throw new ArithmeticException("sorry");
}
	2. void m()throws ArithmeticException{
	//method code
	}

3. Checked exception cannot be propagated using throw only.	3. Checked exception can be propagated with throws.
4. Throw is followed by an instance.	4. Throw is followed by a class.
5. Throw is used within the method.	5. Throws is used with the method signature.
6. You cannot throw multiple exceptions.	6. You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

finally

- ***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.
- ***finally*** block 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.

finally

```
// Demonstrate finally.
class FinallyDemo {
        // Throw an exception out of the method.
        static void procA( ) {
                try {
                        System.out.println("inside procA");
                        throw new RuntimeException("demo");
                } finally {
                        System.out.println("procA's finally");
                }
        }

        // Return from within a try block.
        static void procB( ) {
                try {
                        System.out.println("inside procB");
                        return;
                } finally {
                        System.out.println("procB's finally");
                }
        }

        // Execute a try block normally.
        static void procC( ) {
                try {
                        System.out.println("inside procC");
                } finally {
                        System.out.println("procC's finally");
                }
        }

        public static void main(String args[ ]) {
                try {
                        procA( );
                } catch (Exception e) {
                        System.out.println("Exception caught");
                }
                procB( );
                procC( );
        }
}
```

finally

Here is the output generated by the program:

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

```
// Execute a try block normally.
static void procC( ) {
        try {
                System.out.println("inside procC");
        } finally {
                System.out.println("procC's finally");
        }
}

public static void main(String args[ ]) {
        try {
                procA( );
        } catch (Exception e) {
                System.out.println("Exception caught");
        }
        procB( );
        procC( );
}
}
```

final vs. finally vs. finalize

No.	final	finally	finalize
1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.

final vs. finally vs. finalize

```
class FinalExample{
        public static void main(String[ ] args){
                final int x=100;
                x=200;          //Compile Time Error
        }
}
```

```
class FinallyExample{
        public static void main(String[ ] args){
                try{
                        int x=300;
                }catch(Exception e){
                        System.out.println(e);
                }
                finally{
                        System.out.println("finally block is executed");
                }
        }
}
```

```
class FinalizeExample{
        public void finalize( ){
                System.out.println("finalize called");
        }
        public static void main(String[ ] args){
                FinalizeExample f1=new FinalizeExample( );
                FinalizeExample f2=new FinalizeExample( );
                f1=null;
                f2=null;
                System.gc( );
        }
}
```

Java's Built-in Exceptions

- Inside the standard package ***java.lang***, Java defines several exception classes.

- The classes which directly inherit ***Throwable*** class except ***RuntimeException*** and ***Error*** are known as **checked exceptions** e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

- The classes which inherit ***RuntimeException*** are known as **unchecked exceptions** e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

Java's Built-in Exceptions

- Java's Unchecked ***RuntimeException*** Subclasses Defined in ***java.lang***

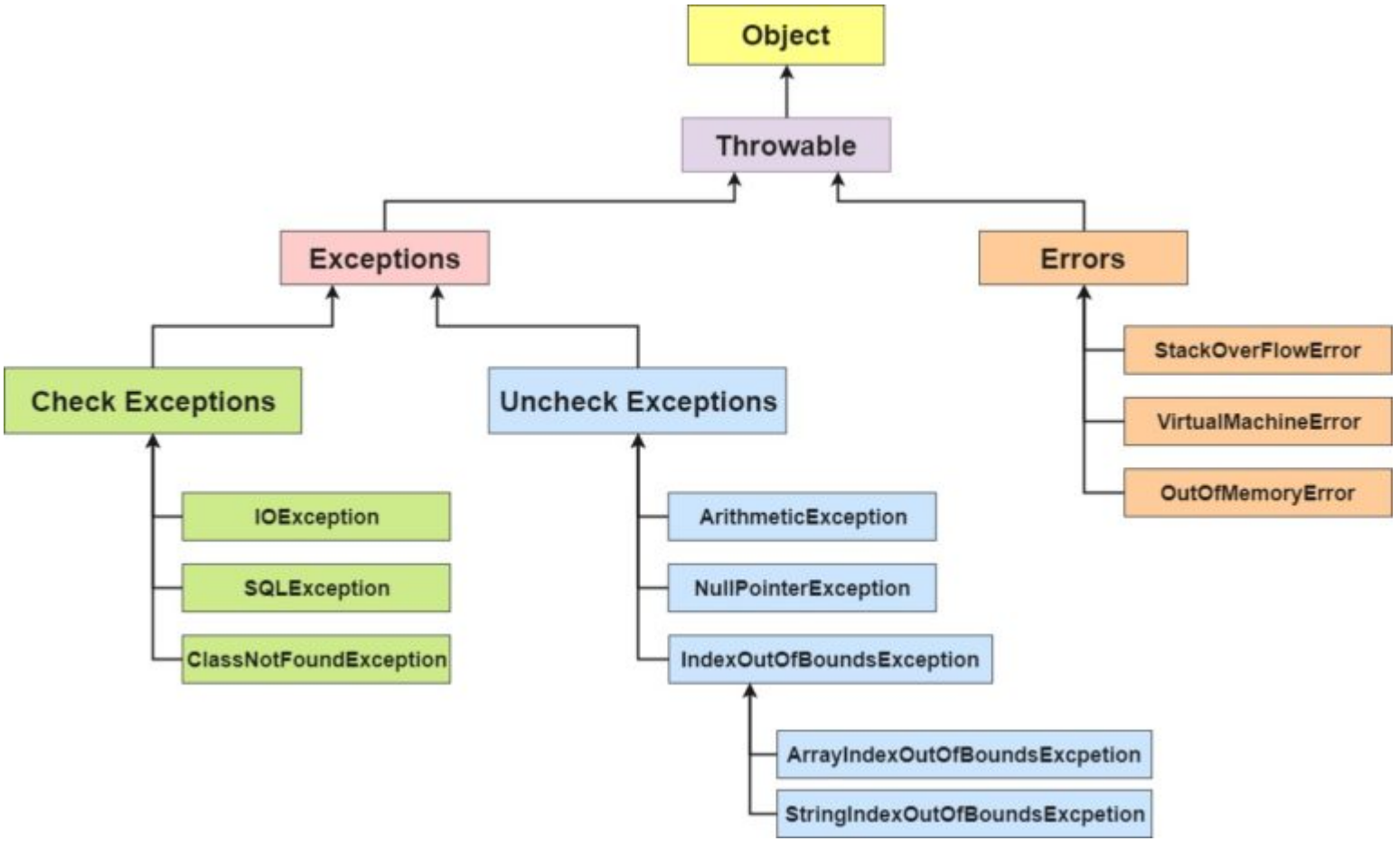
Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made 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.

Java's Built-in Exceptions

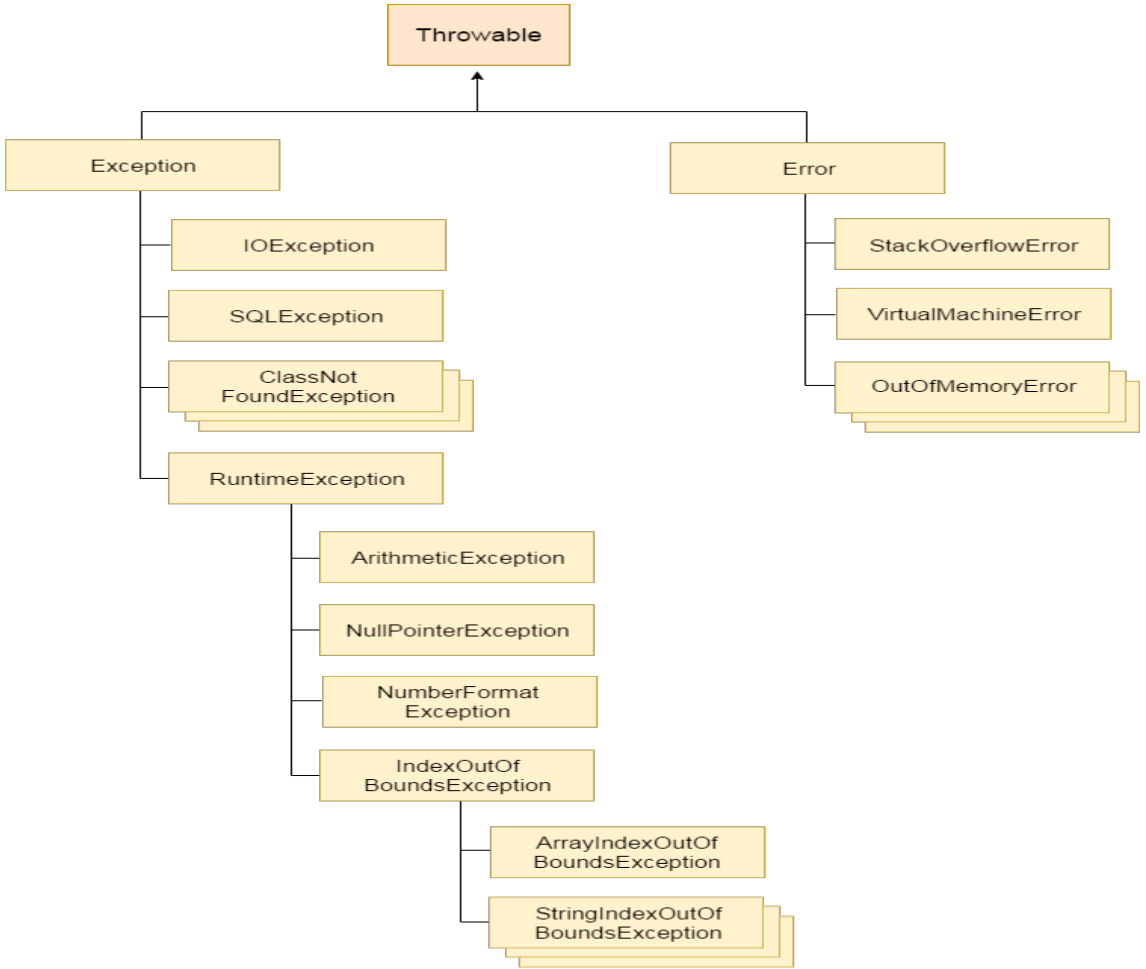
- Java's Checked Exceptions Defined in ***java.lang***

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.

Hierarchy of Java Exception Classes



Hierarchy of Java Exception Classes



Creating your own Exception Subclass

- Sometimes you may 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 ***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:

```
Exception( )
Exception(String msg)

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

] The **chained exception** feature allows you to associate another exception with an exception.

Creating your own Exception Subclass

- The Methods Defined by Throwable

Method	Description
final void addSuppressed(Throwable *exc*)	Adds *exc* to the list of suppressed exceptions associated with the invoking exception. Primarily for use by the **try**-with-resources statement.
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 getLocalizedMessage( )	Returns a localized description of the exception.
String getMessage( )	Returns a description of the exception.
StackTraceElement[ ] getStackTrace( )	Returns an array that contains the stack trace, one element at a time, as an array of **StackTraceElement**. The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The **StackTraceElement** class gives your program access to information about each element in the trace, such as its method name.

Creating your own Exception Subclass

- The Methods Defined by Throwable

final Throwable[ ] getSuppressed( )	Obtains the suppressed exceptions associated with the invoking exception and returns an array that contains the result. Suppressed exceptions are primarily generated by the **try**-with-resources statement.
Throwable initCause(Throwable *causeExc*)	Associates *causeExc* with the invoking exception as a cause of the invoking exception. Returns a reference to the exception.
void printStackTrace( )	Displays the stack trace.
void printStackTrace(PrintStream *stream*)	Sends the stack trace to the specified stream.
void printStackTrace(PrintWriter *stream*)	Sends the stack trace to the specified stream.
void setStackTrace(StackTraceElement *elements*[ ])	Sets the stack trace to the elements passed in *elements*. This method is for specialized applications, not normal use.
String toString( )	Returns a **String** object containing a description of the exception. This method is called by **println( )** when outputting a **Throwable** object.

Creating your own Exception Subclass

```
// This program creates a custom exception type.

class MyException extends Exception {
        private int detail;
        MyException(int a) {
                detail = a;
        }
        public String toString( ) {  //overrides toString( )
                return "MyException[" + detail + "]";
        }
}
class ExceptionDemo {
        static void compute(int a) throws MyException {
                System.out.println("Called compute(" + a + ")");
                if(a > 10)
                        throw new MyException(a);

                System.out.println("Normal exit");
        }
        public static void main(String args[ ]) {
                try {
                        compute(1);
                        compute(20);
                } catch (MyException e) {
                        System.out.println("Caught " + e);
                }
        }
}
```

Here is the output of the program:

```
Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]
```

References

Reference for this topic

- [Book: Java: The Complete Reference, Ninth Edition: Herbert Schildt]
  https://www.amazon.in/Java-Complete-Reference-Herbert-Schildt/dp/0071808558

- [Web: GeeksforGeeks]
  https://www.geeksforgeeks.org/java/

- [Web: Java T Point tutorial]
  https://www.javatpoint.com/java-tutorial