

# Exception handling

Ebey S.Raj

# Exception

2

- An exception is an abnormal condition that arises in a code sequence at run time.
- An exception is a runtime error.

# Exception handling fundamentals

3

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



# Exception handling fundamentals

4

- 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

5

- 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**.
  - Code in the **catch** block can catch this exception and handle it.
  - 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 must be executed after a try block completes is put in a **finally** 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.

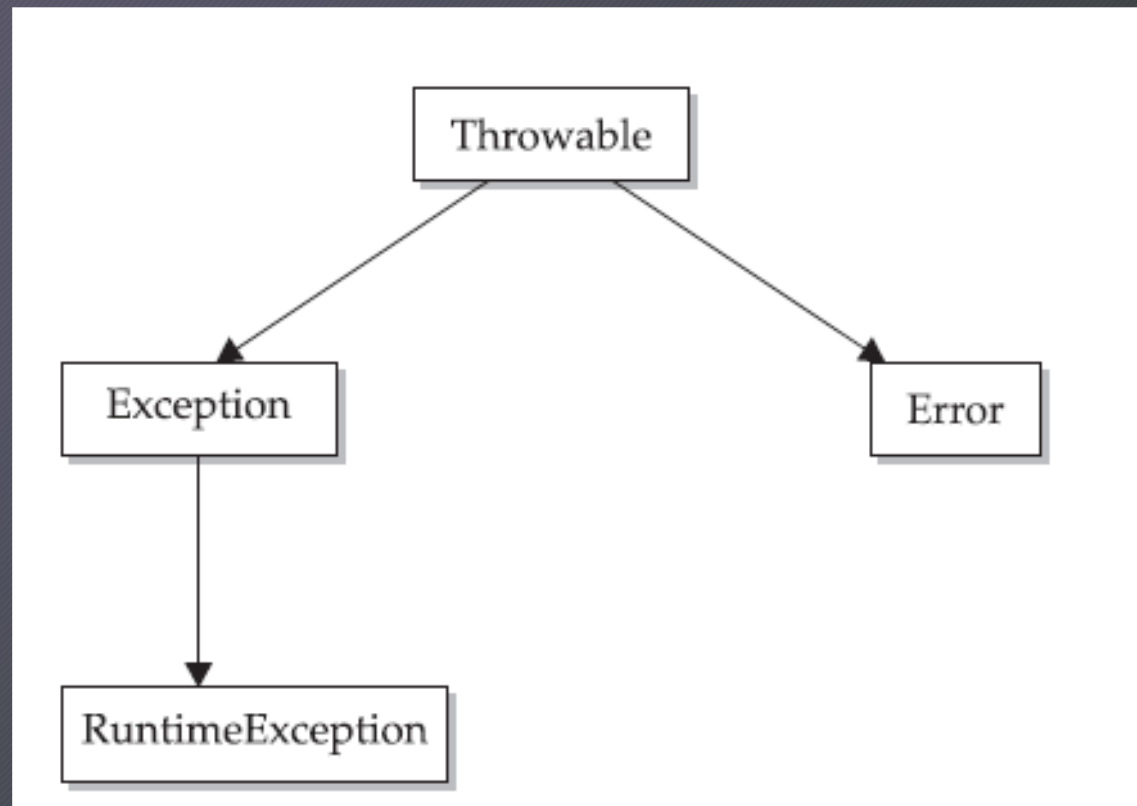
**General form of an exception handling block**



# Exception Types

7

- All exception types are subclasses of the built-in class **Throwable**.
- Throwable has two subclasses.
  - Exception
    - Exception has one subclass called RuntimeException.
  - Error



# Exception Types

8

- Exception
  - Exception class is used for exceptional conditions that user programs should catch.
  - Manually generated exceptions should inherit this class.
  - There is an important subclass of Exception, called RuntimeException.
    - Exceptions of this type are automatically defined for the programs that we write.
    - Eg: Division by zero and Invalid array indexing.



# Exception Types

9

- Error
  - Error defines exceptions that are not expected to be caught under normal circumstances by your program.
  - Errors are used by the Java run-time system to indicate errors having to do with the run-time environment, itself.
  - Eg: Stack overflow
  - These are typically created in response to catastrophic failures that cannot usually be handled by our program.

# Uncaught Exception

10

- Consider the class

```
class Exc0 {  
    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 this exception.
- Once an exception has been thrown, it must be caught by an exception handler and dealt with immediately.
- Since we haven't supplied any exception handlers of our own, the exception is caught by the **default handler** provided by the Java run-time system and terminates the execution of Exc0.

# Uncaught Exception

11

- 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.

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



# Uncaught Exception: Another example

12

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

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

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

# Using try and catch

13

- In the case of default handling of exception by Java runtime system, the program will be terminated by the run time system.
- Handling exception by users provides two benefits
  - It allows you to fix the error.
  - It prevents the program from automatically terminating.

# Using try and catch

14

- To guard against and handle a run-time error,
  - 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.



# Exception Handling using try and catch

15

```
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 divide-by-zero error  
            System.out.println("Division by zero.");  
        }  
        System.out.println("After catch statement.");  
    }  
}
```

## Output

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

# Points to ponder: try and catch

16

- 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
- The statements that are protected by try must be surrounded by curly braces.
- We cannot use try on a single statement.

# try – catch: Another example

Another Example: [HandleError.java](#)

catch clauses resolves the exceptional condition and helps to continue as if the error had never happened.

We can display the description of the exception in a `println( )` statement by simply passing the exception as an argument.

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

## Output

Exception: java.lang.ArithmeticException: / by zero



# Multiple catch clauses

18

- To handle multiple exceptions raised by a single piece of code, we can specify two or more catch clauses.
- Each catch clause catches 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: Example

19

- Multiple Catch Clause Example: [MultipleCatches.java](#)

# Point to ponder

20

- 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.




# Catching subclass exceptions: An example

21

Created by Ebey S.Raj

```
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.");  
        }  
        /* This catch is never reached because  
        ArithmeticException is a subclass of Exception. */  
        catch(ArithmeticException e) { // ERROR – unreachable  
            System.out.println("This is never reached.");  
        }  
    }  
}
```

A subclass must come before its superclass in a series of catch statements. **If not, unreachable code will be created and a compile-time error will result.**



# Catching subclass exceptions: An example

22

Created by Ebey S.Raj

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

**A subclass must come before its superclass in a series of catch statements.**



# Nested try statement

23

- A try statement can be used inside the block of another try block.
- 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 catch Example: NestTry.java](#)



# Nested try statement

24

- Nesting of try statements can occur in less obvious ways when method calls are involved.
  - We can enclose a call to a method within a try block and inside that method we can use another try statement.

[Nested Try catch with methods example: MethNestTry.java](#)

# throw

25

- It is possible 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.

# throw

26

- There are two ways to obtain a Throwable object:
  - creating one with the new operator.
  - using a parameter in a catch clause
- The flow of execution stops immediately after the throw statement.
  - Any subsequent statements are not executed.
- And then it continuously inspects the nearest 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 for a matching catch, and so on.
  - If no matching catch is found, then the **default exception handler** halts the program and prints the stack trace.



# throw: example

27

- [ThrowDemo.java](#)
- Many of Java's built-in run-time exceptions have at least two constructors:
  - one with no parameter and
  - one that takes a string parameter.
    - The argument specifies a string that describes the exception.
    - This string is displayed when the object is used as an argument to `print( )` or `println( )`.

# throws

28

- If a method is capable of causing an exception that it does not handle, it can specify this behavior to the callers by including a **throws** clause in the method's declaration.
  - Callers of the method can guard themselves against that exception.
- 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.

# General form of a method declaration that includes a throws clause

29

```
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.

[Example: ThrowsDemo.java](#)



# finally

30

- **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.
  - 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 clause is also executed just before the method returns.

# finally

31

- **finally** can be useful for closing file handles and freeing up any allocated resources.
- The finally clause is optional.
- Each try statement requires at least one **catch** or a **finally** clause.

[Example: FinallyDemo.java](#)

# Java Built-in Exceptions

32

- Java defines several exception classes inside the standard package **java.lang**.
- The most general of these exceptions are subclasses of the standard type `RuntimeException`.
- Unchecked Exceptions
  - The compiler does not check to see if a method handles or throws these exceptions.
  - The unchecked exceptions defined in `java.lang` are listed in following table.



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.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

## Java Built-in Unchecked Exceptions

# Java Built-in Exceptions

34

- Checked Exceptions
  - Exceptions 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.
  - Following table lists those exceptions defined by java.lang.
- Java defines several other types of exceptions that relate to its various class libraries.

# Java Built-in Checked Exceptions

35

Exception	Meaning
<code>ClassNotFoundException</code>	Class not found.
<code>CloneNotSupportedException</code>	Attempt to clone an object that does not implement the <code>Cloneable</code> interface.
<code>IllegalAccessException</code>	Access to a class is denied.
<code>InstantiationException</code>	Attempt to create an object of an abstract class or interface.
<code>InterruptedException</code>	One thread has been interrupted by another thread.
<code>NoSuchFieldException</code>	A requested field does not exist.
<code>NoSuchMethodException</code>	A requested method does not exist.
<code>ReflectiveOperationException</code>	Superclass of reflection-related exceptions. (Added by JDK 7.)



# Creating our own Exception classes

36

- To create your own exception types to handle situations specific to your applications
  - Just define a subclass of **Exception**
  - The Exception class does not define any methods of its own.
  - It inherits those methods provided by Throwable. We can override one or more of these methods in exception classes that you create.

# Constructors of Exception class

37

```
Exception( )  
Exception(String msg)
```

- The first form creates an exception that has no description.
- The second form lets you specify a description of the exception.
- When an exception is created with description, it is better to override `toString( )`. By overriding `toString( )`, we can make our own cleaner output.

# Methods provided by Throwable

38

Method	Description
final void addSuppressed(Throwable <i>exc</i> )	Adds <i>exc</i> to the list of suppressed exceptions associated with the invoking exception. Primarily for use by the new <b>try-with-resources</b> statement. (Added by JDK 7.)
Throwable fillInStackTrace( )	Returns a <b>Throwable</b> 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, <b>null</b> 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 <b>StackTraceElement</b> . 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 <b>StackTraceElement</b> class gives your program access to information about each element in the trace, such as its method name.



# Methods provided by Throwable

39

Method	Description
<code>final Throwable[ ] getSuppressed( )</code>	Obtains the suppressed exceptions associated with the invoking exception and returns an array that contains the result. Suppressed exceptions are primarily generated by the new <b>try-with-resources</b> statement. (Added by JDK 7.)
<code>Throwable initCause(Throwable <i>causeExc</i>)</code>	Associates <i>causeExc</i> with the invoking exception as a cause of the invoking exception. Returns a reference to the exception.
<code>void printStackTrace( )</code>	Displays the stack trace.
<code>void printStackTrace(PrintStream <i>stream</i>)</code>	Sends the stack trace to the specified stream.
<code>void printStackTrace(PrintWriter <i>stream</i>)</code>	Sends the stack trace to the specified stream.
<code>void setStackTrace(StackTraceElement <i>elements</i>[ ])</code>	Sets the stack trace to the elements passed in <i>elements</i> . This method is for specialized applications, not normal use.
<code>String toString( )</code>	Returns a <b>String</b> object containing a description of the exception. This method is called by <b>println( )</b> when outputting a <b>Throwable</b> object.

# Creating our own Exception classes

40

- Creating our own Exception: MyException

[ExceptionDemo.java](#)

# References

41

- Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.