09 Exception Handling

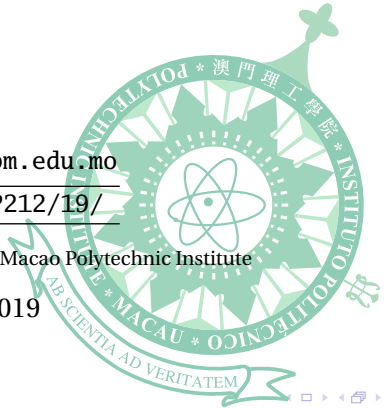
Instructor: Ke Wei（柯韋）

A319 Ext. 6452 wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP212/19/

Bachelor of Science in Computing, School of Applied Sciences, Macao Polytechnic Institute

October 31 (November 12), 2019



Outline

Exception Handling Overview

- An exception is a *runtime* error.
- A program that does not provide code for *catching* and *handling* exceptions will terminate abnormally, and may cause serious problems.
- Exceptions occur for various reasons.
  - The user may enter an invalid input, or
  - the program may attempt to open a file that doesn't exist, or
  - the network connection may hang up, or
  - the program may attempt to access an out-of-bounds array element.
- An *ArrayIndexOutOfBoundsException* occurs if you access an element past the end of an array.
- A *NullPointerException* occurs when you invoke a method on a reference variable with null value.
- Text I/O operations may also throw exceptions.

An Example: *InputMismatchException*

```
1  public class ExceptionDemo {
2      public static void main(String[] args) {
3          Scanner scanner = new Scanner(System.in);
4          System.out.print("Enter an integer: ");
5          int number = scanner.nextInt();
6          System.out.println("The number entered is "+number);
7      }
8  }
```

If an exception occurs on this line, the rest of the lines in the method are skipped and the program is terminated.

```
Enter an integer: 4.8
Exception in thread "main" java.util.InputMismatchException
at java.util.Scanner.throwFor(Unknown Source)
at java.util.Scanner.next(Unknown Source)
at java.util.Scanner.nextInt(Unknown Source)
at java.util.Scanner.nextInt(Unknown Source)
at ExceptionDemo.main(ExceptionDemo.java:6)
```

Catching the Exception

Java allows the program to catch and process exceptions.

```
1           public static void main(String[] args) {
2               Scanner scanner = new Scanner(System.in);
3               for ( ; ; ) {
4                   try {
5                       System.out.print("Enter an integer: ");
6                       int number = scanner.nextInt();
7                       System.out.println("The number entered is "+number);
8                       break;
9                   } catch ( InputMismatchException ex ) {
10                      System.out.println("An integer is required");
11                      scanner.nextLine();     // discard current input
12                  }
13              }
14          }
```

If an exception occurs on this line, the rest of the lines in the try block are skipped and the control is transferred to the catch block.

The Try/Catch Statement

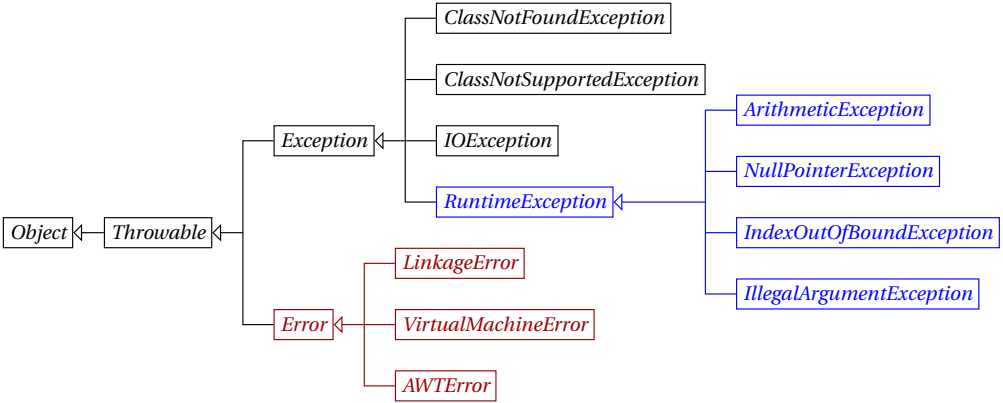
- A `try`/`catch` statement consists of a `try` block and one or more `catch` blocks.

  `try` { *statements* } `catch` (*an exception parameter*) { *statements* }
  `catch` (*an exception parameter*) { *statements* } ...

- A `try` block begins with the keyword `try` followed by a block of statements in braces.
- A `try` block contains the statements that *might* throw exceptions.
- A `catch` block begins with the keyword `catch` followed by an exception parameter in parentheses and a block of statements for handling the exception in braces.
- When a statement in a `try` block throws an exception, the rest of the statements in the `try` block are skipped and control is transferred to the `catch` block.

Exceptions and Exception Classes

- A Java exception is an instance of a class derived from *Throwable*.
- You can create your own exception classes by extending *Throwable* or a subclass of *Throwable*.



Classification of Exception Classes

The exception classes can be classified into three major types: system errors, exceptions, and runtime exceptions.

- System errors are thrown by the JVM and represented in the *Error* class. The *Error* class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.
- Exceptions are represented in the *Exception* class, which describes errors caused by your program and by external circumstances. These errors can be caught and handled by your program.
- Runtime exceptions are represented in the *RuntimeException* class, which describes programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors. Runtime exceptions are generally thrown by the JVM.

Checked and Unchecked Exceptions

- *RuntimeException*, *Error*, and their subclasses are known as *unchecked exceptions*.
- All other exceptions are known as *checked exceptions*, meaning that the compiler forces the programmer to check and deal with them.
- In most cases, unchecked exceptions reflect programming logic errors that are not recoverable. These logic errors should be corrected in the program.

Declaring and Throwing Exceptions

Java's exception-handling model is based on three operations: declaring an exception, throwing an exception, and catching an exception.

- Every method must state the types of checked exceptions it might throw. This is known as declaring exceptions.

```
public void myMethod()
    throws IOException, MyException { ... }
```

- A program that detects an error can create an instance of an appropriate exception class and throw it. This is known as throwing an exception.

```
MyException ex = new MyException("Wrong Case");
throw ex;
```

Or,

```
throw new MyException("Wrong Case");
```

Catching Exceptions

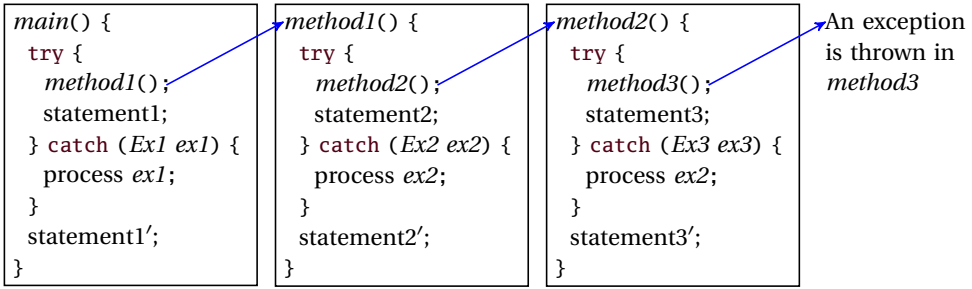
When an exception is thrown, it can be caught and handled in a `try-catch` block, as follows:

```
try {
    statements // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
    handler for exception1
}
catch (Exception2 exVar2) {
    handler for exception2
}
...
catch (ExceptionN exVar3) {
    handler for exceptionN
}
```

If no exceptions arise during the execution of the `try` block, the `catch` blocks are skipped.

Exception Propagation

Suppose the *main* method invokes *method1*, *method1* invokes *method2*, *method2* invokes *method3*, and an exception occurs in *method3*.



Call Stack:

Exception Inheritance

- Various exception classes can be derived from a common superclass.
- If a catch block catches exception objects of a superclass, it can catch all the exception objects of the subclasses of that superclass.
- The order in which exceptions are specified in `catch` blocks is important.
- A compilation error will result if a `catch` block for a superclass type appears before a `catch` block for a subclass type.

```
try {
  ...
} catch ( Exception ex ) {
  ...
} catch ( RuntimeException ex ) {
  ...
}
```

```
try {
  ...
} catch ( RuntimeException ex ) {
  ...
} catch ( Exception ex ) {
  ...
}
```

Getting Information from Exceptions

- An exception object contains information about the exception.
- The methods in the *Throwable* class return the information.
  - *String getMessage*() — returns the detail message string of `this` throwable.
  - *String toString*() — returns a string combining the exception class name and the detail message of `this` throwable.
  - `void` *printStackTrace*() — prints `this` throwable and its backtrace to the console.
  - *StackTraceElement*[] *getStackTrace*() — returns an array of stack trace element pertaining to `this` throwable.

```
catch (Exception ex) {
    for ( StackTraceElement te : ex.getStackTrace() )
        System.out.println(   "method " + te.getMethodName()
                            + "(" + te.getClassName()
                            + ":" + te.getLineNumber() + ")");
}
```

Example: Declaring and Throwing Exceptions

```
1  public class CircleWithException {
2      private double radius;
3      private static int numberOfObjects = 0;
4      public CircleWithException() { this(1.0); }
5      public CircleWithException(double newRadius) {
6          setRadius(newRadius);
7          numberOfObjects++;
8      }
9      public double getRadius() { return radius; }
10     public void setRadius(double newRadius)
11         throws IllegalArgumentException {
12         if ( newRadius >= 0 ) radius = newRadius;
13         else throw new IllegalArgumentException("negative radius");
14     }
15     public static int getNumberOfObjects() { return numberOfObjects; }
16     public double getArea() { return radius * radius * 3.14159; }
17 }
```

Example: Catching Exceptions

```
1  public class TestCircleWithException {
2    public static void main(String[] args) {
3      try {
4        CircleWithException a = new CircleWithException(5);
5        CircleWithException b = new CircleWithException(-5);
6        CircleWithException c = new CircleWithException(0);
7      }
8      catch (IllegalArgumentException ex) {
9        System.out.println(ex);
10     }
11
12     System.out.println("Number of objects created: "
13                        + CircleWithException.getNumberOfObjects());
14   }
15 }
```

Rethrowing Exceptions

Java allows an exception handler to rethrow the exception if the handler cannot process the exception or the handler simply wants to let its caller be notified of the exception.

```
try {
  statements
}
catch ( MyException ex ) {
  perform operations before exits
  throw ex;
}
```

The statement `throw ex` rethrows the exception so that other handlers get a chance to process the exception *ex*.

The Finally Clause

- Occasionally, you may want some code to be executed regardless of whether an exception occurs or is caught.
- Java has a `finally` clause that can be used to accomplish this objective.

```
try { statements }
catch ( MyException ex ) { handling ex }
finally { final statements }
```

- The code in the `finally` block is executed under all circumstances, regardless of whether an exception occurs in the `try` block or is caught.
- The `finally` block executes even if there is a `return` statement prior to reaching the `finally` block.
- The `catch` block may be omitted when the `finally` clause is used.

Example: Releasing Resources in Finally Blocks

A common use of the `finally` clause is in I/O programming. To ensure that a file is closed under all circumstances, you may place a file closing statement in the `finally` block.

```
1  public class FinallyDemo {
2      public static void main(String[] args) {
3          java.io.PrintWriter output = null;
4          try {
5              output = new java.io.Printwriter("text.txt");
6              output.println("Welcome to Java");
7          } catch ( java.io.IOException ex ) {
8              ex.printStackTrace();
9          } finally {
10             if (output != null) output.close();
11         }
12     }
13 }
```

When to Use Exceptions

- If you want the exception to be processed by the method caller, you should create an exception object and throw it.
- If you can handle the exception in the method where it occurs, there is no need to throw or use exceptions.
- In general, common exceptions that may occur in multiple classes in a project are candidates for exception classes.

```
try { System.out.println(refVar.toString()); }
catch ( NullPointerException ex ) { System.out.println("refVar is null"); }
```

is better replaced by

```
if ( refVar != null ) System.out.println(refVar.toString());
else System.out.println("refVar is null");
```

- The point is not to abuse exception handling as a way to deal with a simple logic test.