# Exception Handling

# What are Exceptions ?

- We don't live in a perfect world.
  - there are so many unexpected situations that cause our programs to behave different from our expectations.
- Exceptions refer to
  - situations where programs cannot behave normally.
  - any abnormal situation can lead to a incorrect program behavior.
- Examples of exceptions
  - Happen to divide integer by zero ➔ it may halt the program
  - Non numeric character is entered when an integer is expected.
  - No available memory space when memory allocation is requested by new.
  - No such file when a program tries to open a file.

## What are Exceptions ?

- To prevent a program from abnormal behavior, we need to check every statement that could cause exceptions.
- What exceptions should be considered in the following code ?

```
double badCode(int a[], int y) {
    x = a[y] ;
    z = a[x] ;
    return 1 / z ;
}
```

  - The array *a* may point to null.
  - The index of *y* may not be within the valid range: 0 .. a.length-1
  - The index of *x* may not be within the valid range: 0 .. a.length-1
  - The value of *z* may be zero.

## The Traditional Approach to Handling Exceptions

- To prevent a program from abnormal behavior, we need to check every statement that could cause exceptions.

```
double badCode(int a[], int y) {
    if ( a == null ) { System.out.println("null array") ; return -1.0; }
    if (y < 0 || y >= a.length) {
        System.out.printf("Array index %d is not valid%n", y); return -2.0 ;
    }
    int x = a[y] ;
    if (x < 0 || x >= a.length) {
        System.out.printf("Array index %d is not valid%n", x); return -3.0 ;
    }
    int z = a[x] ;
    if (z == 0) {
        System.out.println("Error: Denominator is 0"); return -4.0 ;
    }
    return 1 / z ;
}
```

## Problems with the Traditional Approach

- Problem #1: Too much overhead for the exception handling codes
- Problem #2: exception handling codes are interspersed with normal codes.
- Problem #3: artificial codes are used to notify exceptional situations.

```
double badCode(int a[], int y) {
    if ( a == null ) { System.out.println("null array") ; return -1.0; }
    if (y < 0 || y >= a.length) {
        System.out.printf("Array index %d is not valid%n", y); return -2.0 ;
    }
    int x = a[y] ;
    if (x < 0 || x >= a.length) {
        System.out.printf("Array index %d is not valid%n", x); return -3.0 ;
    }
    int z = a[x] ;
    if (z == 0) {
        System.out.println("Error: Denominator is 0"); return -4.0 ;
    }
    return 1 / z ;
}
```

# Exception Handling in Java

- With try/catch block, we can separate the error checking codes from the normal code.
- If that exception is not been handled then the program will terminate.
- Exception handling in Java is similar to that in C++.

```
statements;
try {
    error-prone code...;
}
catch (Exception-type1 e1) {
    code for dealing with e1 exception
}
catch (Exception-type2 e2) {
    code for dealing with e2 exception
}
more-statements;
```

Try block wraps the error-prone code

If an exception (error) occurs anywhere in the code inside the try block, the catch block is executed immediately

After the catch block (the catch handler) has finished, execution continues after the catch block

## badCode with Exception Handling

```
public class ExceptionHandling_1 {
  private static double badCode(int a[], int y) throws Exception {
      try {
          int x = a[y] ; // NullPointerException
          int z = a[x] ;
          return 1 / z ;
      }
      catch ( Exception e ) {
          System.out.println("Exception occurred: " + e) ;
          throw e ;
      }
  }
  public static void main(String[] args)  {
      try { badCode(null, 10) ; }
      catch ( Exception e) { System.out.println("badCode failed") ; }
  }
}
```

```
Exception occurred: java.lang.NullPointerException
badCode failed
```

## badCode with Exception Handling

```
public class ExceptionHandling_2 {
   private static double badCode(int a[], int y) throws Exception {
       try {
           int x = a[y] ; // ArrayIndexOutOfBoundsException
           int z = a[x] ;
           return 1 / z ;
       }
       catch ( Exception e ) {
           System.out.println("Exception occurred: " + e) ;
           throw e ;
       }
   }
   public static void main(String[] args)  {
       try { int[] a = {0, 1, 2} ; badCode(a, 3); }
       catch ( Exception e) { System.out.println("badCode failed") ; }
   }
}
```

```
Exception occurred: java.lang.ArrayIndexOutOfBoundsException:
               Index 3 out of bounds for length 3
badCode failed
```

# Standard Exceptions in Java

- Many useful exceptions are defined as subclasses of *RuntimeException* class in Java.
- They belong to java.lang package, not requiring importing

| Exception | Description |
|---|---|
| **NullPointerException** | Thrown when an application attempts to use null in a case where an object is required |
| **ArithmeticException** | Thrown when an exceptional arithmetic condition has occurred; e.g.) divide by zero |
| **ClassCastException** | Thrown when an application attempts an illegal cast; for example<br>`Object x = new Integer(0);`<br>`System.out.println((String)x);` |
| **ArrayIndexOutOfBounds Exception** | Thrown to indicate that an array has been accessed with an illegal index |

## Trace with Exception: printStackTrace()

```
01: public class ExceptionHandling_3 {
02:   private static double badCode(int a[], int y) throws Exception {
03:       try {
04:           int x = a[y] ; // throws ArrayIndexOutOfBoundsException
05:           int z = a[x] ;
06:           return 1 / z ;
07:       }
08:       catch ( Exception e ) {
09:           System.out.println("Exception occurred: " + e) ;
10:           e.printStackTrace();
11:           throw e ;
12:       }
13:   }
14:   public static void main(String[] args)  {
15:        try { int[] a = {0, 1, 2} ; badCode(a, 3); }
16:       catch ( Exception e) { System.out.println("badCode failed") ; }
17:   }
18: }
```

You can trace the source of the exception by printStackTrace()

```
Exception occurred: java.lang.ArrayIndexOutOfBoundsException: 3
java.lang.ArrayIndexOutOfBoundsException: 3
        at ExceptionHandling_3.badCode(ExceptionHandling_3.java:4)
        at ExceptionHandling_3.main(ExceptionHandling_3.java:15)
badCode failed
```

# Multiple Catch Handlers

```
import java.util.Scanner;
public class MultipleCatchHandlers {
 public static void main(String[] args) {
    try {
      Scanner scanner = new Scanner(System.in) ;
      int x = scanner.nextInt() ; // can throw java.util.InputMismatchException
      int[] a = {-1, 0, 1, 2} ;
      int y = a[x] ; // can throw ArrayIndexOutOfBoundsException
      int z = a[y]/a[y]; // can throw ArrayIndexOutOfBoundsException and ArithmeticException
    }
    catch (ArithmeticException e) {
      System.out.println("Arithmetic exception took place: " + e) ;
    }
    catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Array index is invalid: " + e) ;
    }
    catch (java.util.InputMismatchException e) {
      System.out.println("The given string cannot be converted into an integer: " + e) ;
    }
 }
}
```

Multiple catch blocks can be provided to handle different types of exceptions.

```
    ...
    try {
       Scanner scanner = new Scanner(System.in) ;
       int x = scanner.nextInt() ;              // (3) java.util.InputMismatchException
       int[] a = {-1, 0, 1, 2} ;
       int y = a[x] ;
       int z = a[y] / a[x] ;          // (1) ArrayIndexOutOfBoundsException, (2) ArithmeticException
    }
    catch (ArithmeticException e) {
       System.out.println("Arithmetic exception took place: " + e) ;
    }
    catch (ArrayIndexOutOfBoundsException e) {
       System.out.println("Array index is invalid: " + e) ;
    }
    catch (java.util.InputMismatchException e) {
       System.out.println("The given string cannot be converted into an integer: " + e) ;
    }
  }
}
```

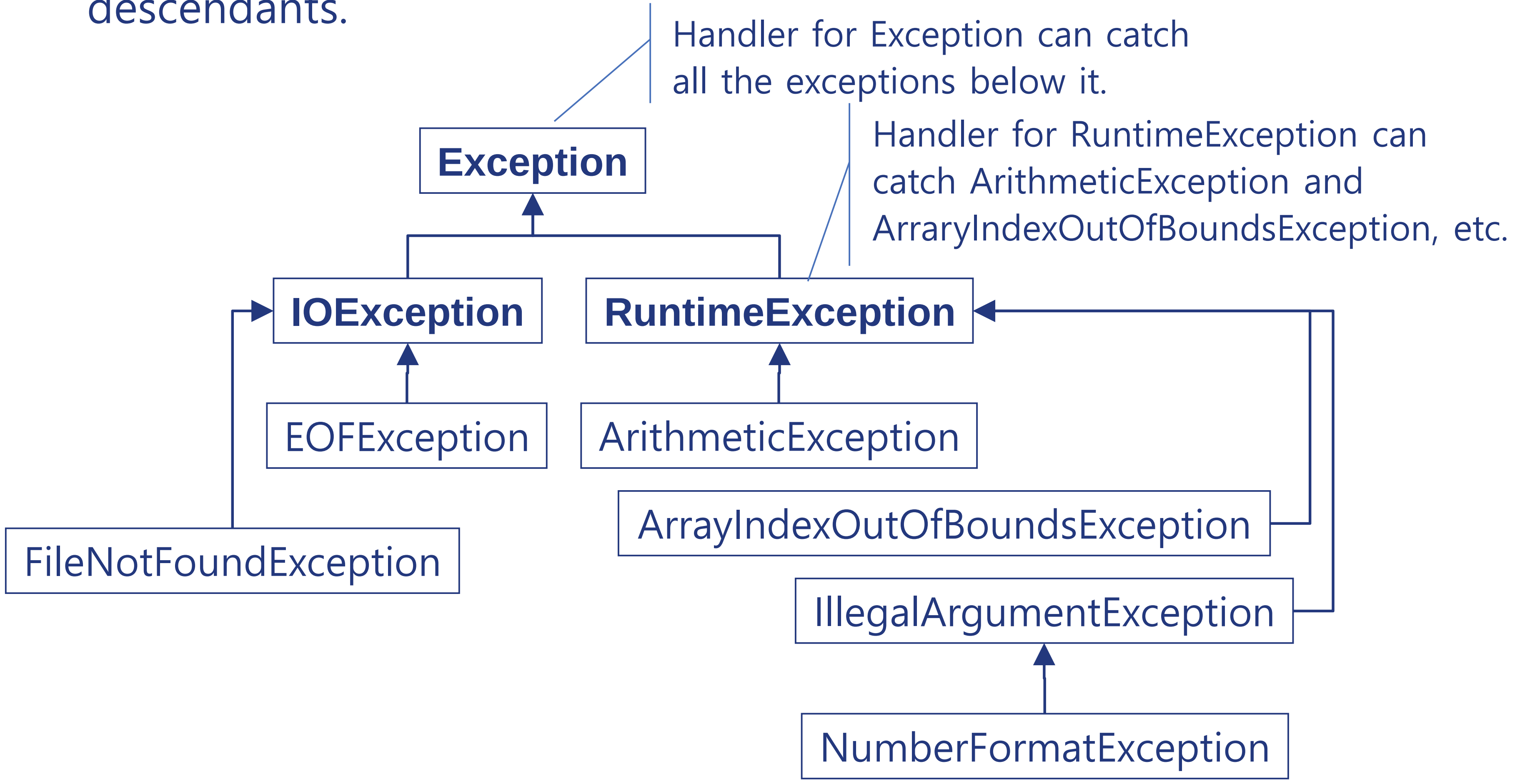
```
0
Array index is invalid: java.lang.ArrayIndexOutOfBoundsException: -1
```

```
1
Arithmetic exception took place: java.lang.ArithmeticException: / by zero
```

```
abc
The given string cannot be converted into an integer: java.util.InputMismatchException
```

# Catch Handler for Multiple Exceptions

- Catch handler can catch all the exceptions of its own type and its descendants.


Handler for Exception can catch all the exceptions below it.
Handler for RuntimeException can catch ArithmeticException and ArraryIndexOutOfBoundsException, etc.
Exception
IOException
RuntimeException
EOFException
ArithmeticException
ArrayIndexOutOfBoundsException
FileNotFoundException
IllegalArgumentException
NumberFormatException

```
import java.io.*;
public class CatchingMultipleExceptions {
 public static void main(String[] args) {
    try {
       FileReader fin = new FileReader("employee.txt") ; // can throw FileNotFoundException
       BufferedReader in = new BufferedReader(fin) ;

       int i1 = Integer.parseInt(in.readLine()) ; // can throw NumberFormatException
       int i2 = Integer.parseInt(in.readLine()) ; // can throw NumberFormatException
       System.out.printf("%d %d%n ", i1, i2) ;

       int[] a = {-10, 0, 10, 20} ;
       System.out.println(a[i1] / a[i2]) ;
       // can throw ArrayIndexOutOfBoundsException and ArithmeticException
    }
    catch ( IOException e) { // can catch java.io.FileNotFoundException
       System.out.println(e) ;
    }
    catch (RuntimeException e) {
       // can catch ArrayIndexOutOfBoundsException, ArithmeticException, and NumberFormatException
       System.out.println(e) ;
    }
 }
}
```

```
2 1
 java.lang.ArithmeticException: / by zero
```

```
0 4
 java.lang.ArrayIndexOutOfBoundsException: 4
```

# The Most General Hander

```
catch (Exception e) {
      ...
}
```

can catch any exception because **Exception** is the superclass of all exception types

```
import java.io.*;
public class CatchingMultipleExceptions {
 public static void main(String[] args) {
    try {
       ...
    }
    catch (Exception e) { System.out.println(e) ; }
    catch ( IOException e) { System.out.println(e) ;  }
    catch (RuntimeException e) { System.out.println(e) ; }
 }
}
```

```
Exception in thread "main" java.lang.Error: Unresolved compilation problems:
      Unreachable catch block for IOException.
      It is already handled by the catch block for Exception
      Unreachable catch block for RuntimeException.
      It is already handled by the catch block for Exception
```

## The finally Block

❖ The finally block *always* executes

```
public class FinallyBlock {
    public static void main(String[] args) {
        String[] array = {"First", "Second", "Third"};
        List<String> list = new ArrayList<>(Arrays.asList(array));
        PrintWriter out = null;
        try {
            System.err.println("Entering try statement");
            out = new PrintWriter( new FileWriter("OutFile.txt"));
            for (int i = 0; i < 10; i++) out.println("Value at: " + i + " = " + list.get(i));
        } catch (IndexOutOfBoundsException e) {
            System.err.println("Caught IndexOutOfBoundsException: " + e.getMessage());
        } catch (IOException e) {
            System.err.println("Caught IOException: " + e.getMessage());
        } finally {
            if (out != null) { System.err.println("Closing PrintWriter"); out.close(); }
            else { System.err.println("PrintWriter not open"); }
        }
    }
}
```

```
Entering try statement
Caught IndexOutOfBoundsException: Index 3 out of bounds for length 3
Closing PrintWriter
```

## Closing a File with Finally Block

- You can use a finally block to ensure that a resource is closed regardless of whether the try statement completes normally or abruptly.

```
public static String readFirstLine(String path) throws IOException {
      BufferedReader br = null;
      String line = "";
      try {
          br = new BufferedReader(new FileReader(path));
          line = br.readLine();
      } finally {
          if ( br != null ) br.close(); // may throw exception
      }
      return line;
   }
```

## The try-with-resources Statement (Since Java 7)

- The try-with-resources statement ensures that each resource is closed at the end of the statement.
- A resource: any object that implements java.lang.AutoCloseable can be used as a resource.

```
public static String readFirstLine(String path) throws IOException {
   String line = "";
   try ( BufferedReader br = new BufferedReader(new FileReader(path)) )
   {
     line = br.readLine();
   }
   return line;
}
```

## User-defined Exceptions: definitions

- Programmer can define new classes of exceptions by extending, commonly, the RuntimeException class.

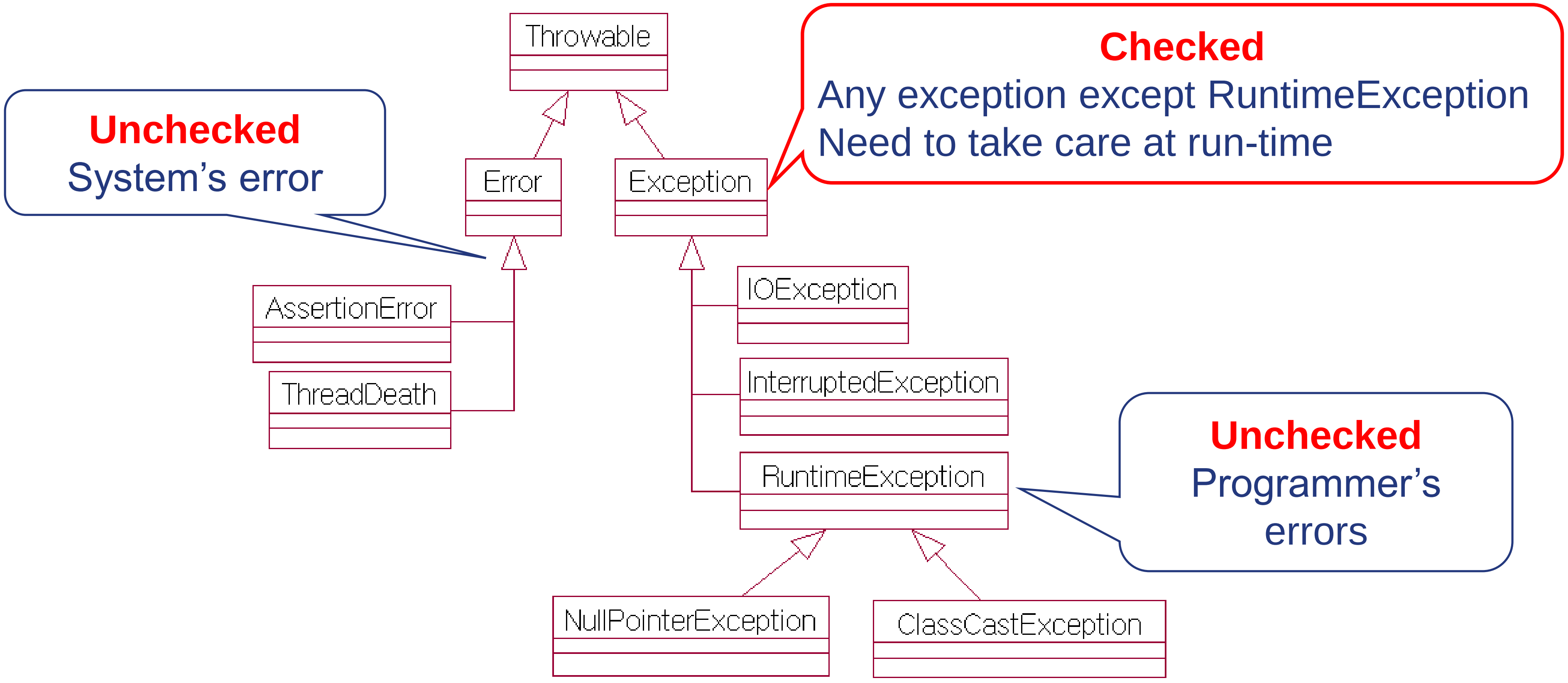
```
class MyDivideByZeroException extends RuntimeException {
    public String toString() {
        return "Zero denominator used" ;
    }
}

class MyArrayOutOfBoundsException extends RuntimeException {
    private final int invalidIndex ;

    public MyArrayOutOfBoundsException(int invalidIndex) {
        this.invalidIndex = invalidIndex ;
    }
    public String toString() {
        return String.format("Invalid Array Index: %d", invalidIndex) ;
    }
}
```

# User-defined Exceptions: Throwing and Catching

```
public class UserDefinedException {
  private static double badCode(int a[], int y) {
    if ( y < 0 || y >= a.length ) throw new MyArrayOutOfBoundsException(y) ;
    int x = a[y] ;

    if ( x == 0 ) throw new MyDivideByZeroException() ;
    return 1 / x ;
  }

  public static void main(String[] args) {
    try {
        int[] a = {0, 1, 2} ;
        badCode(a, 5) ; // throw MyArrayOutOfBoundsException(y)
        // badCode(a, 0) ; // throw MyDivideByZeroException()
    }
    catch (MyDivideByZeroException e ) { System.out.println(e) ; }
    catch ( MyArrayOutOfBoundsException e) { System.out.println(e) ; }
  }
}
```

# Types of Exceptions

- Exceptions are classified into **checked** and **unchecked**



# How to handle checked exceptions

- For checked exceptions, you should catch them

```
class MyRangeException extends Exception { // checked
   public String toString() { return “MyRangeException" ; }
}
class MyNumberException extends RuntimeException { // unchecked
   public String toString() { return " MyNumberException " ; }
}
public class CheckedException {
   public static void write() {
        try {
             FileReader fin = new FileReader("employee.txt") ;
             throw new MyNumberException() ;
             throw new MyRangeException() ;
        }
        catch ( IOException e) { … } // catching checked exception
        catch ( MyRangeException e) { … } // catching checked exception
   }
}
```

## How to handle checked exceptions

- Or, you should rethrow, so that the caller can handle them

```
class MyRangeException extends Exception { // checked exception
   public String toString() { return “MyRangeException" ; }
}
class MyNumberException extends RuntimeException { // unchecked exception
   public String toString() { return " MyNumberException " ; }
}
public class CheckedException {
   private static void write() throws IOException, MyRangeException {
            FileReader fin = new FileReader("employee.txt") ;
            throw new MyNumberException() ;
            throw new MyRangeException() ;
   }
   public static void main(String[] args) throws IOException, MyRangeException {
       write() ;
   }
}
```

# How to handle checked exceptions

- If there are no catch or rethrows for checked exceptions, compiler will issue an error message

```
public class CheckedException {
   private static void write() {
            FileReader fin = new FileReader("employee.txt") ;
            // unhandled exception type FileNotFoundException
            throw new MyNumberException() ;
            throw new MyRangeException() ;
            // unhandled exception type MyRangeException
   }
}
```

```
public class CheckedException {
   private static void write() throws IOException, MyRangeException {
            FileReader fin = new FileReader("employee.txt") ;
            throw new MyNumberException() ;
            throw new MyRangeException() ;
   }
   public static void main(String[] args)  {
       write() ;
   }
}
```

## Not Handling an Exception

- If a method raises an exception and it does *not* have a catch block or throws declaration then…
  - ***checked exception***: the compiler will reject your code at compile time: You should catch or rethrows it !
  - ***unchecked exception:*** the program will terminate at runtime

# Not Handling an Unchecked Exception

- Unhandled unchecked exceptions

```
1: class MyNumberException extends RuntimeException {
2:      public String toString() { return "MyNumberException" ; }
3: }
4: public class UncheckedException {
5:      public static void main(String[] args) {
6:           write() ; // not handle MyNumberException
7:      }
8:      public static void write()  {
9:           throw new MyNumberException() ;
10:     }
   }
```

```
Exception in thread "main" MyNumberException
        at UncheckedException.write(UncheckedException.java:9)
        at UncheckedException.main(UncheckedException.java:6)
```

## Stack Trace Information

- If no handler is called, then the system prints a stack trace as the program terminates
  - it is a list of the called methods that are waiting to return when the exception occurred
  - very useful for debugging/testing

- The stack trace can also be printed by calling exception.printStackTrace()

```
01: public class UsingStackTrace {
02:  public static void main( String args[] ) {
03:     try { method1(); }
04:     catch ( Exception exception ) {
05:         System.err.println( exception.getMessage() + "₩n" );
06:         exception.printStackTrace();
07:         StackTraceElement[] traceElements = exception.getStackTrace();
08:         System.out.println(  "₩nStack trace from getStackTrace:" );
09:         System.out.println( "Class₩t₩tFile₩t₩t₩tLine₩tMethod" );
10:         for ( int i = 0; i < traceElements.length; i++ ) {
11:            StackTraceElement element = traceElements[ i ];
12:            System.out.print( element.getClassName() + "₩t" );
13:            System.out.print( element.getFileName() + "₩t" );
14:            System.out.print( element.getLineNumber() + "₩t" );
15:            System.out.print( element.getMethodName() + "₩n" );
16:         }
17:      }
18:   }
19:   public static void method1() throws Exception { method2(); }
20:   public static void method2() throws Exception { method3(); }
21:   public static void method3() throws Exception { throw new Exception( "Exception
 thrown in method3" ); }
}
```

The compiler will reject the program at compile time if the throws are not included because Exception is checked exception

```
Exception thrown in method3

java.lang.Exception: Exception thrown in method3
        at UsingStackTrace.method3(UsingStackTrace.java:21)
        at UsingStackTrace.method2(UsingStackTrace.java:20)
        at UsingStackTrace.method1(UsingStackTrace.java:19)
        at UsingStackTrace.main(UsingStackTrace.java:3)

Stack trace from getStackTrace:
Class                   File                    Line    Method
UsingStackTrace UsingStackTrace.java    21      method3
UsingStackTrace UsingStackTrace.java    20      method2
UsingStackTrace UsingStackTrace.java    19      method1
UsingStackTrace UsingStackTrace.java    3       main
```

# Advantages of Exceptions (1/3)

❖ Separating Error-Handling Code from "Regular" Code

```
int readFile {
    initialize errorCode = 0;

    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
    …
```

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
       doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

# Advantages of Exceptions (2/3)

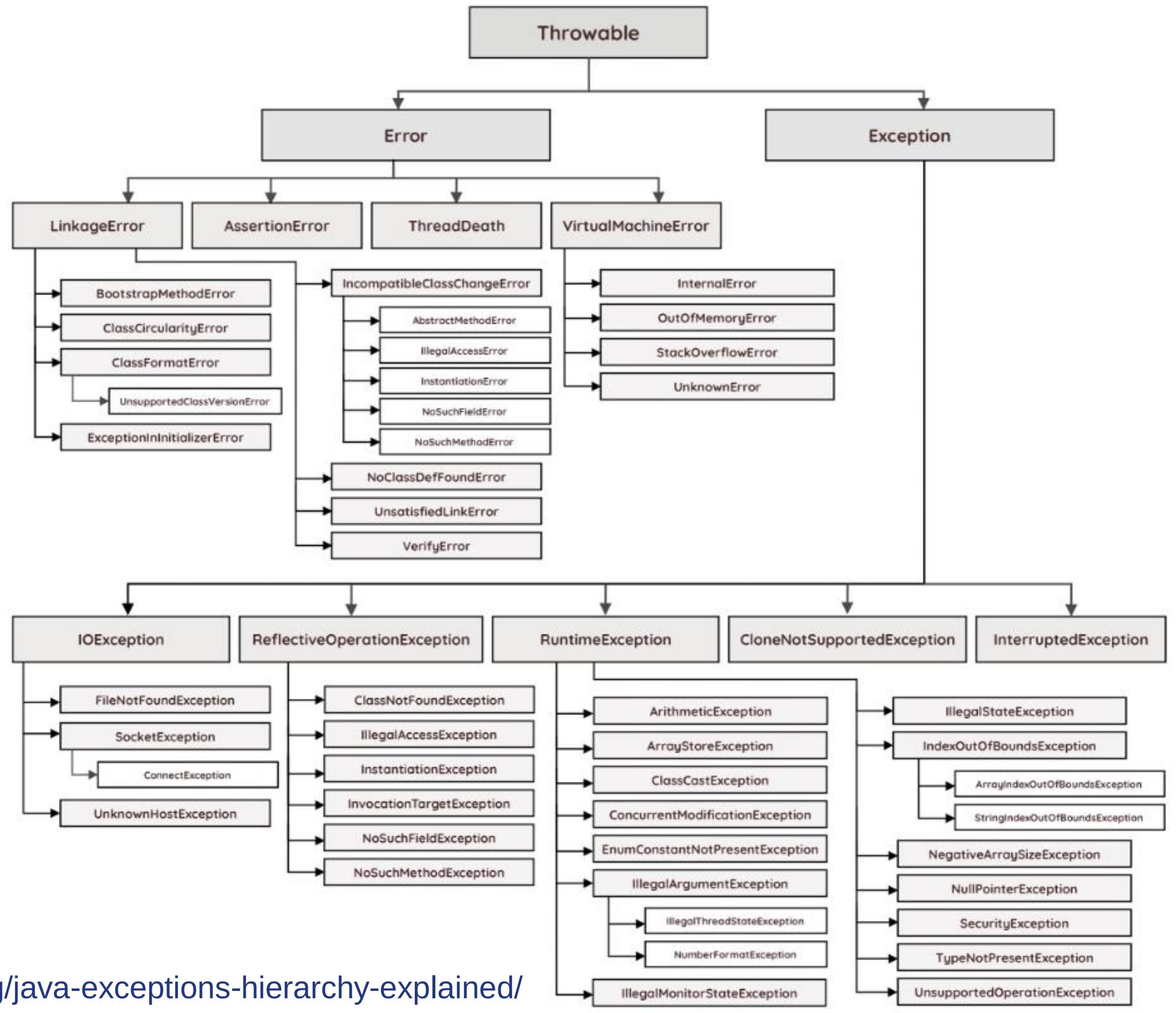
## Propagating Errors Up the Call Stack

```
method1 {
   int error;
   error = call method2;
   if (error)
      doErrorProcessing;
   else
      proceed;
}

int method2 {
   int error;
   error = call method3;
   if (error)
      return error;
   else
      proceed;
}

int method3 {
   int error;
   error = call readFile;
   if (error)
      return error;
   else
      proceed;
}
```

```
// Suppose also that method1 is the only
method interested in the errors that might
occur within readFile

method1 {
    try {
        call method2;
    } catch (exception e) {
        doErrorProcessing;
    }
}

method2 throws exception {
    call method3;
}

method3 throws exception {
    call readFile;
}
```

# Advantages of Exceptions (3/3)

❖ Grouping and Differentiating Error Types



https://rollbar.com/blog/java-exceptions-hierarchy-explained/

# Best Practices to Handle Exception

- Remember "Throw early catch late" principle
- Never use exceptions for flow control in your program
- Clean Up Resources in a Finally Block or Use a Try-With-Resource Statement
  - Always clean up after handling the exception
- Prefer Specific Exceptions
  - Declare the specific checked exceptions that your method can throw
- Document the Exceptions You Specify
  - Document all exceptions in the application with javadoc
- Throw Exceptions With Descriptive Messages
- Don't Catch Throwable
  - Never catch Throwable class
- Don't Ignore Exceptions
  - Never swallow the exception in catch block
- Don't Log and Throw
  - Either log the exception or throw it but never do the both
- Wrap the Exception Without Consuming it
- Never throw any exception from finally block
- Don't use printStackTrace() statement or similar methods

https://dzone.com/articles/9-best-practices-to-handle-exceptions-in-java
https://howtodoinjava.com/best-practices/java-exception-handling-best-practices/

## Using Assertions

- With assertions, you can use assertions to check some conditions.

```
1: public class AssertTest {
2:   public static void main(String args[]) {
3:       assert args.length > 0 : "₩nusage: java AssertTest <args>" ;
4:       int i = 0;
5:       do {
6:           System.out.println(args[i]);
7:           i++;
8:       } while (i < args.length);
9:   }
  }
```

```
Exception in thread "main" java.lang.AssertionError:
usage: java AssertTest <args>
        at AssertTest.main(AssertTest.java:3)
```

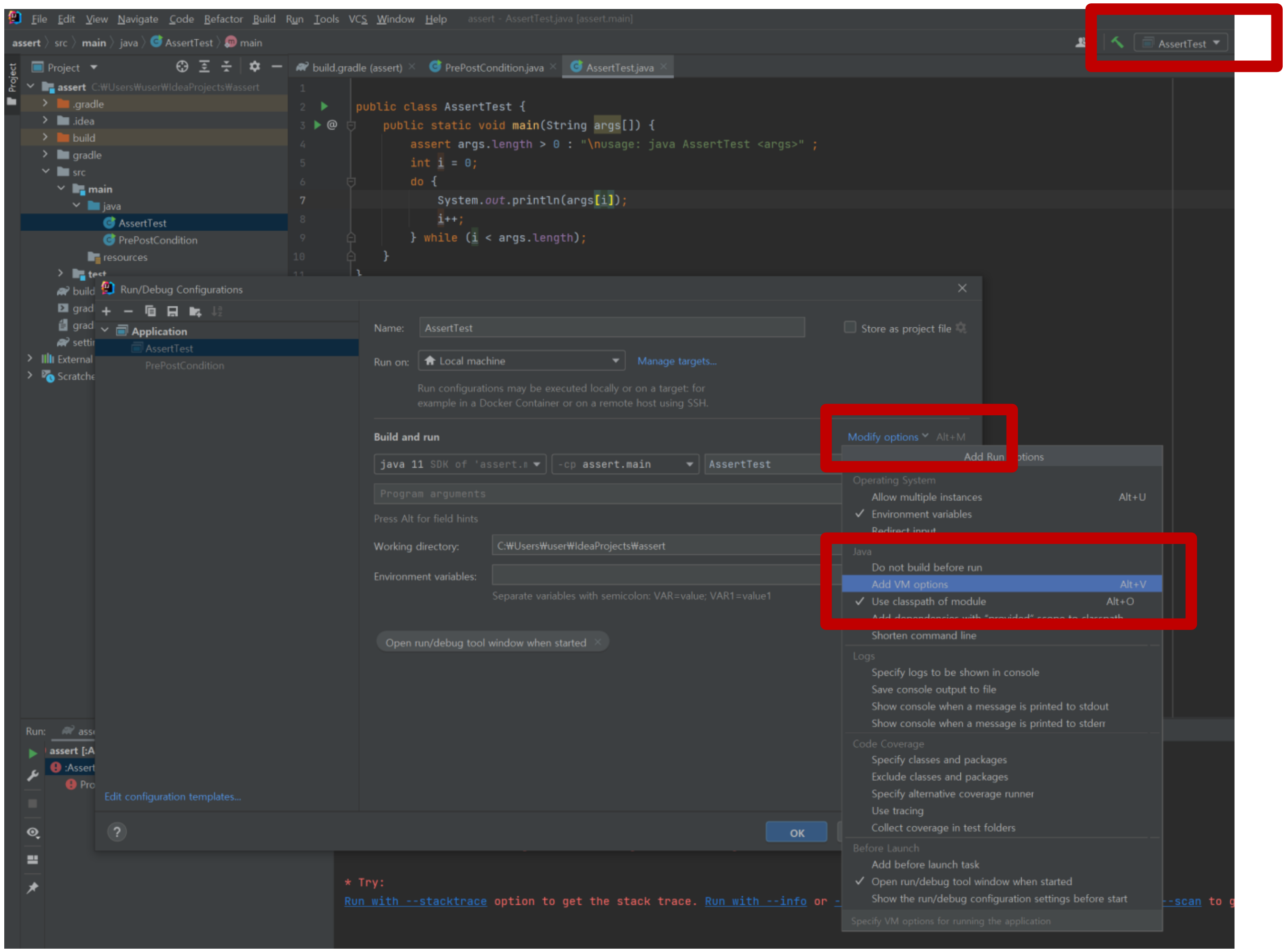
# Using Assertions

- Usages for assert
  - assert Condition ;
  - assert Condition : Expression ;

  - where:
    - Condition is a boolean expression.
    - Expression is an expression that has a value. (It cannot be an invocation of a method that is declared void.)

- Assertions can be enabled and disabled by –ea option
  - java –ea AssertTest
  - java –da AssertTest

# Enabling and Disabling assertions in IntelliJ



File Edit View Navigate Code Refactor Build Run Tools VCS Window Help
assert - AssertTest.java [assert.main]
assert 〉 src 〉 main 〉 java 〉 AssertTest 〉 main
AssertTest
build.gradle (assert)
PrePostCondition.java
AssertTest.java
Project
Run/Debug Configurations
Application
AssertTest
PrePostCondition
Edit configuration templates...
Modify options Alt+M
Add Run Options
Operating System
Allow multiple instances Alt+U
✓ Environment variables
Redirect input
Java
Do not build before run
Add VM options Alt+V
✓ Use classpath of module Alt+O
Add dependencies with "provided" scope to classpath
Shorten command line
Logs
Specify logs to be shown in console
Save console output to file
Show console when a message is printed to stdout
Show console when a message is printed to stderr
Code Coverage
Specify classes and packages
Exclude classes and packages
Specify alternative coverage runner
Use tracing
Collect coverage in test folders
Before Launch
Add before launch task
✓ Open run/debug tool window when started
Show the run/debug configuration settings before start
Specify VM options for running the application
OK

```java
public class AssertTest {
    public static void main(String args[]) {
        assert args.length > 0 : "\nusage: java AssertTest <args>" ;
        int i = 0;
        do {
            System.out.println(args[i]);
            i++;
        } while (i < args.length);
    }
}
```

Name: AssertTest

☐ Store as project file

Run on: Local machine

Manage targets...

Run configurations may be executed locally or on a target: for example in a Docker Container or on a remote host using SSH.

**Build and run**

`java 11 SDK of 'assert.m` `-cp assert.main` `AssertTest`

Program arguments

Press Alt for field hints

Working directory: C:₩Users₩user₩IdeaProjects₩assert

Environment variables:

Separate variables with semicolon: VAR=value; VAR1=value1

Open run/debug tool window when started

```
* Try:
Run with --stacktrace option to get the stack trace. Run with --info or ... --scan to g
```

## Enabling and Disabling assertions: Comparison

- When assertions are enabled; java –ea AssertTest

```
Exception in thread "main" java.lang.AssertionError:
usage: java AssertTest <args>
        at AssertTest.main(AssertTest.java:3)
```

- When assertions are disabled; java –da AssertTest

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
        Index 0 out of bounds for length 0
        at AssertTest.main(AssertTest.java:6)
```

## Checking Pre/post condition by assertions

- Precondition: conditions that should be satisfied before the execution
- Postcondition: conditions that should be satisfied after the execution

```
public class PrePostCondition {
    public static void sort(int []a) {
        assert a != null ;                            // precondition
        for ( int i = 0 ; i < a.length - 1; i ++ )
            for ( int j = i + 1 ; j < a.length - 1; j ++ )
                if ( a[i] < a[j] ) {
                        int t = a[i] ; a[i] = a[j] ; a[j] = t ;
                }
        for ( int i = 0 ; i < a.length - 1; i ++ )    // postcondition
            assert a[i] >= a[i+1] ;
    }
    public static void main(String[] args) {
        int[] a = {0, 25, 20, 50} ;
        sort(a) ;
    }
}
```

```
Exception in thread "main" java.lang.AssertionError
            at PrePostCondition.sort(PrePostCondition.java:12)
            at PrePostCondition.main(PrePostCondition.java:16)
```

## Q&A