Exceptions

Lecture 10-1

Don't be afraid to give up the good and go for great.

Outline

- Motivation
- Intro to Exceptions
- Exception Hierarchy
  - Checked Exceptions
  - Unchecked Exceptions
- Exception Handling
  - Catching exceptions
  - Throwing exceptions
- Guidelines

Motivation

- Errors can happen during program execution.
  - Access index out of the array bounds
  - Null reference
  - Integer division by zero

```
int[] intArr = new int[60];
intArr[61] = 78;
```

```
Object mObject = null;
mObject.getClass();
```

```
int nom = 9; int denom = 0;
nom = nom/denom;
```

Motivation

- Some errors are beyond the control of application programmers.
  - Network goes down
  - Out of computer memory
  - Missing input file
  - Full disk

main() method

```
File file = new File("/dev/null/NonExistentPath");
try { InputStream inputStream = new FileInputStream( file ); }
catch (FileNotFoundException fe) { fe.printStackTrace(); }
```

How to Resolve Errors?

- A few naive methods.
  - Ignore the error.
  - Immediately terminate the program.
  - Error codes as a function return value.
    - Uncertain whether the function caller checks the error code.
- They are not generalizable and lacks robustness.

Exception

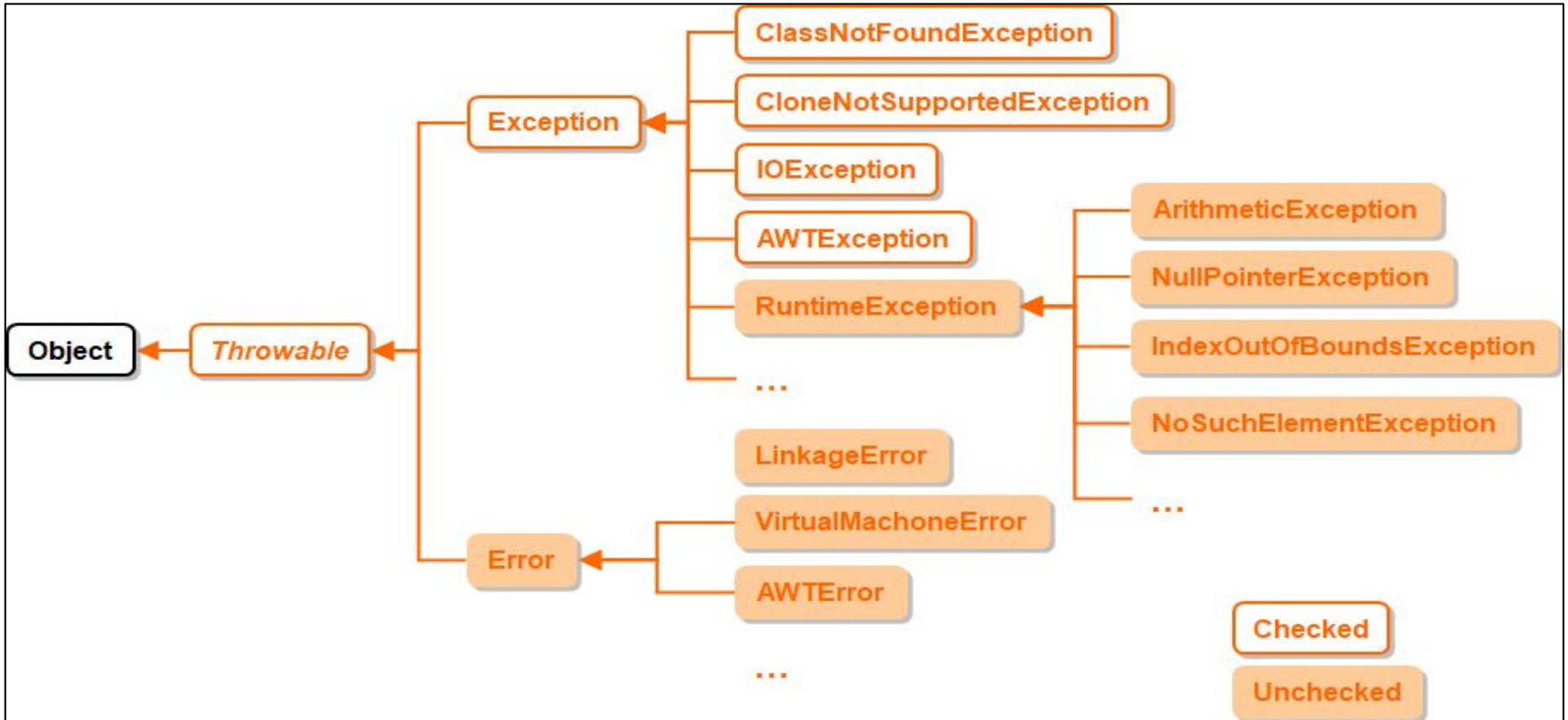
- Java's approach of handling errors.
- An abnormal event signaled by JVM which violates the semantic constraints of Java.
- Provides information about the errors.

Exception

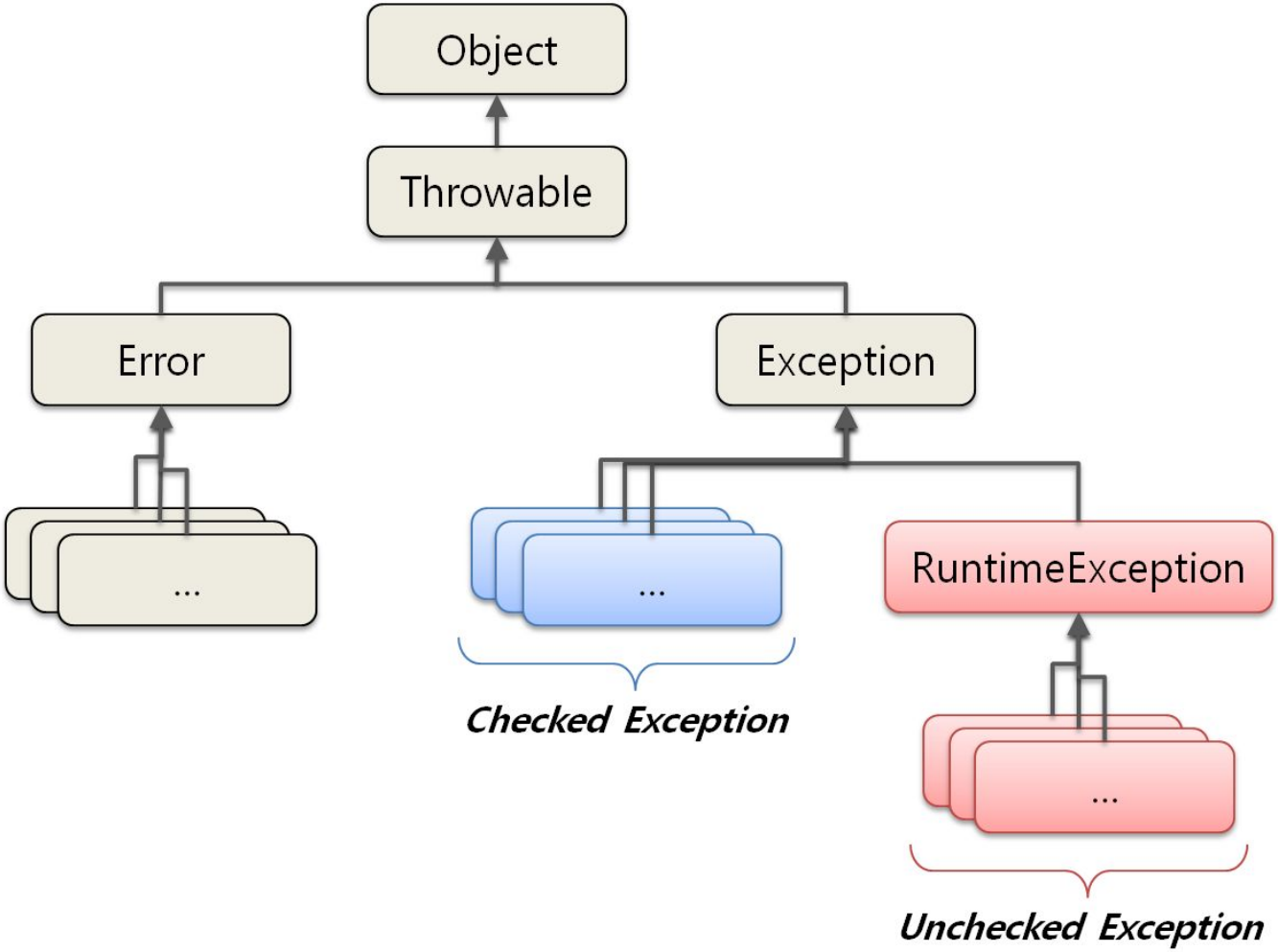
- **Throwing an Exception**: If an error occurs while executing a statement, JVM creates an **exception object**, and halt the normal program flow.
  - The exception object contains a lot of debugging information such as method hierarchy, line number exception occurred, type of exception, etc.
- **Handling an Exception**: JVM tries to find an *Exception Handler*, which is the block of code that can process the exception object.

Exception Object and Hierarchy

- Exception is represented by an instance of Throwable class or one of its subclasses.
- There are **checked** and **unchecked** exceptions.



Simplified View



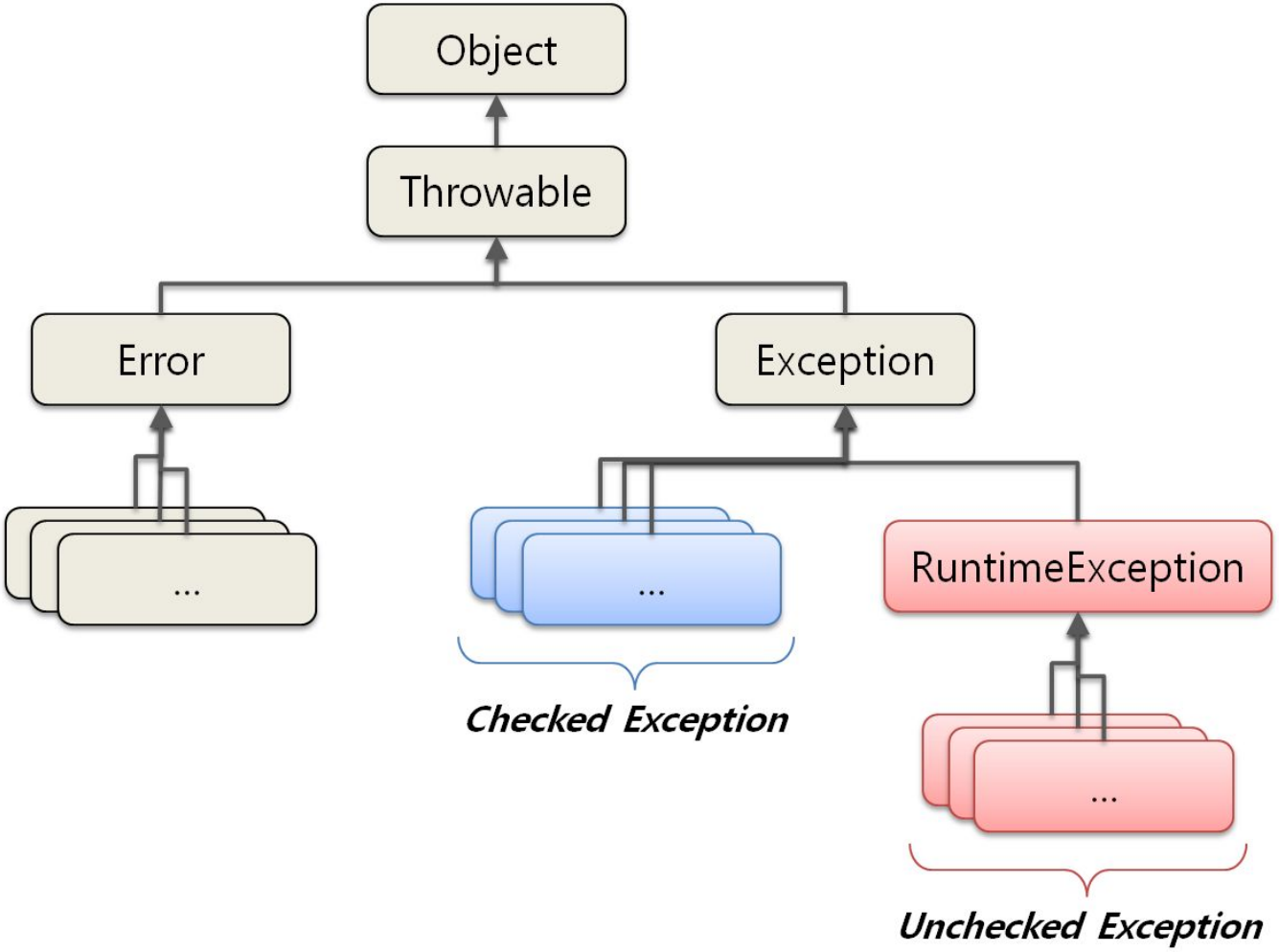
Checked Exceptions

- They are subclasses of the Exception class.
  - Note: RuntimeException is handled differently even though it is a subclass of the Exception class..
- Checked exceptions are evaluated at compile time.
- They must be explicitly handled by the programmers. Otherwise, compilation errors will be generated.

Example Checked Exceptions

- IOException
  - Exceptions related to I/O
    - FileNotFoundException, EOFException, etc.
- ClassNotFoundException
  - Trying to reference the class which corresponding .class file is not found
- InstantiationException
  - Trying to initiate abstract class or interface
- SQLException
  - Trying to access or query database

Unchecked Exceptions



Unchecked Exceptions

- **RuntimeException**: The subclass of Exception that can be thrown during the normal operation of JVM.
  - All runtime exceptions inherits from the RuntimeException class.
  - It is optional to catch these RuntimeExceptions, but still recommended to catch them.
- **Error**: The subclass of Throwable that indicates serious problems that an application should not try to catch.

Example RuntimeExceptions

- NullPointerException
  - Trying to reference from null value
- IndexOutOfBoundsException
  - Accessing index larger than the size of array / string
- IllegalArgumentException
  - Abnormal or incorrect arguments for method
- ArithmeticException
  - Integer division by zero, etc.
- ClassCastException
  - Wrong cast to a class that is not in an inheritance relationship.

Example Errors

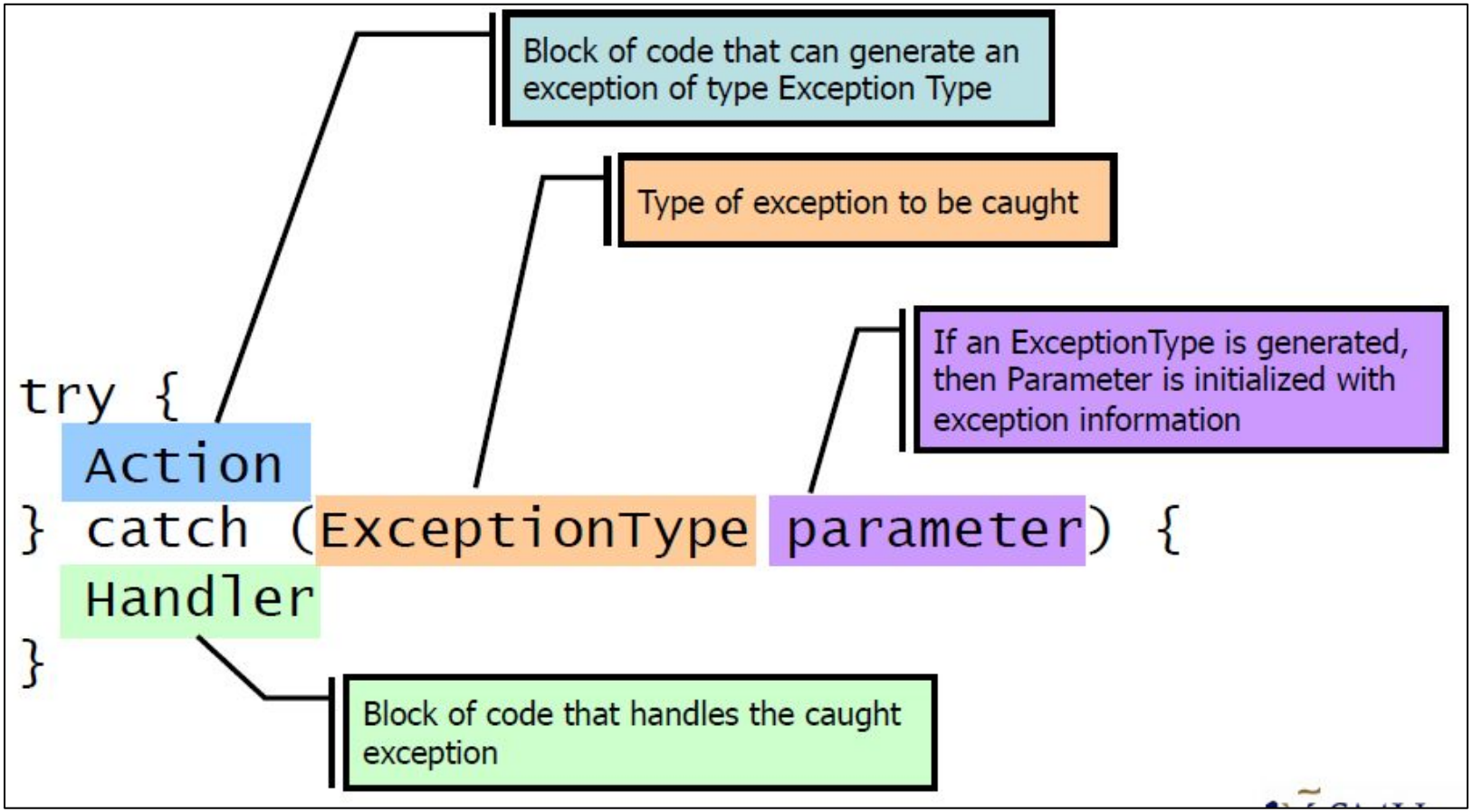
- StackOverflowError
  - Trying to call a new method when the method call stack is full
- VirtualMachineError
  - Internal error or resource limitations in JVM
- OutOfMemoryError
  - Trying to allocate new memory when Machine memory is full

Handling Exceptions

- To simply speak, there are two different methods to handle exceptions.
  - Method 1: Use try-catch to actively handle exceptions.
  - Method 2: Rethrowing exceptions to defer handling.
- Exception handling in Java isn't an easy topic.
  - Beginners find it hard to understand and even experienced developers can spend hours discussing how and which Java exceptions should be thrown or handled.

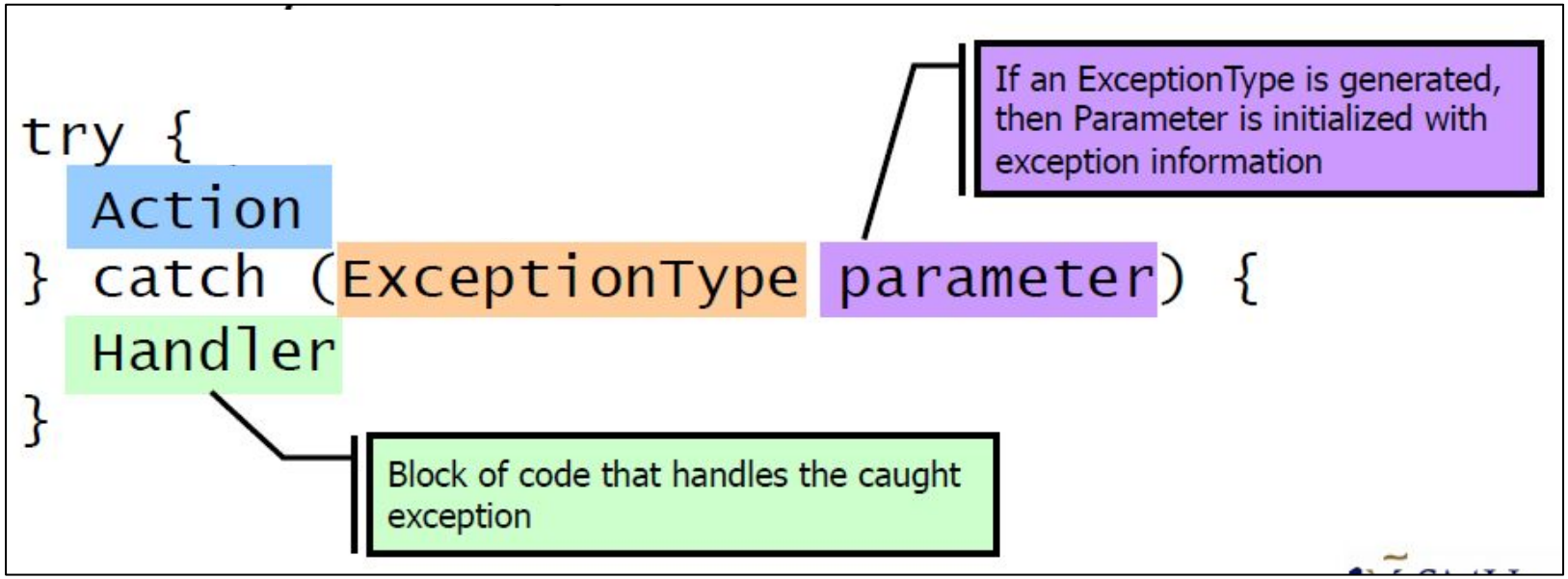
Try-catch Statement

- Try block: A guarded region monitored for errors.



Try-catch Statement

- Catch block
  - A block of code to be executed when a code in the try block throws an expected error.
  - Exception type could be used to monitor all the possible exceptions



What to do in Catch Statement?

- Leave necessary messages for debugging.
  - Use getMessage(), toString(), printStackTrace()
  - Add all the necessary contexts (e.g., values of variables)
- Resolve the error in an appropriate way, if possible.

```
static void test(){
    Object mObject = null;
    System.out.println( mObject.getClass());
}
```

main() method

```
try {
    test();
} catch (Exception e){
    e.printStackTrace();
}
```

Output

```
java.lang.NullPointerException
    at main.test(main.java:16)
    at main.main(main.java:9)
```

Multiple Exception Catch

- A try-catch statement can have several catches, to handle a code block throwing different exceptions.
  - Each catch statement is inspected in order, to find the first matching catch statement.

```
Scanner scn = new Scanner(System.in);
try {
    int n = Integer.parseInt(scn.nextLine());
    if (99 % n == 0)
    System.out.println(n + " divides 99");
} catch (ArithmeticException ex) {
    ex.printStackTrace();
} catch (Exception ex) { // Catches NumberFormatException
    ex.printStackTrace();
}
```

Finally

- Codes to execute whether exception happened or not, after all the exception handling is done.
  - To set things back to its original state after the exception (close files, network connections, etc.)
  - Prevent resource leaks

```
try {
     //statements that may cause an exception
} catch ( … ) {
     //error handling code
} finally {
    //statements to be executed
}
```

Finally Example 1

```
String fileName = "/dev/null/NonExistentPath";
Scanner file = null;
try {
     file = new Scanner(new File(fileName));
} catch (FileNotFoundException e) {
    System.out.println(
         "FileNotFoundException Caught");
    e.printStackTrace();
} finally{ // Prevent Resource Leaks
   if (file != null) {  file.close(); }
}
```

Finally Example 2 (1/2)

```
static int thrower(String s) {
  try {
    if (s.equals("divide")) {
      int i = 0;
      return i / i;
    }
    if (s.equals("null")) {
      s = null;
      return s.length();
    }
    return 0;
  } finally {
    System.out.println( "[thrower(\"" + s + "\") done]");
  }
}
```

Finally Example 2 (2/2)

main() method

```
Scanner scn = new Scanner(System.in);
try {
  thrower(scn.nextLine());
  System.out.println("Didn't throw an exception");
} catch (Exception e) {
  System.out.println(
    e.getClass() + " with : " + e.getMessage());
}
```

Output

divide	[thrower("divide") done] class java.lang.ArithmeticException with : / by zero
null	[thrower("null") done] class java.lang.NullPointerException with : null

Try-with-resources Statement

- try statement that declares one or more resources.
  - A *resource* is an object implementing *java.lang.AutoCloseable* that must be closed after the program is finished with it.
  - The try-with-resources statement ensures that each resource is closed at the end of the statement.

```
try (MyResource resource = new MyResource()) {
     // Do something
} // Resource is automatically closed after this
block ends
```

Try-with-resources Statement

```
static String readLine(String path) throws IOException {
  try (BufferedReader br =
          new BufferedReader(new FileReader(path))) {
        return br.readLine();
  } // BufferedReader implements AutoCloseable
}
```

```
// If we implement with try-finally
static String readLineF(String path) throws IOException {
   BufferedReader br =
      new BufferedReader(new FileReader(path));
   try {
     return br.readLine();
   } finally {
     if (br != null) br.close();
   }
}
```

Try-with-resources Statement

- Multiple resources can be enclosed with try-with-resources.

```
try (resource1; resource2; resource3) {
     // Do something
}
```

```
try (
   BufferedReader br = new BufferedReader(
                           new FileReader("in.txt"));
   BufferedWriter bw = new BufferedWriter(
                           new FileWriter("out.txt"))
) {
   bw.write(br.readLine());
} catch (IOException e) { }
```

Throwing Exception

- Throws exceptions explicitly with throw statements.
- Constructor call of exception class.
  - Arguments could be feeded to constructor to carry the information of the exception.

```
static <T1,T2> ArrayList<Pair<T1,T2>>
zipList(List<T1> l1, List<T2> l2) {
  if (l1.size() != l2.size()) {
    throw new IllegalArgumentException("Length of two
Lists not same");
  }
...
```

Rethrowing an Exception

- Exception handling is deferred to the caller method.

```
static void putfirstline(String fname) throws Exception {
    BufferedReader in;
    try {
        in = new BufferedReader(new FileReader(fname));
    } catch (FileNotFoundException e) {
        System.err.println("Could not open " + fname);
        throw e;
    } catch(Exception e) {
        throw e;
    }
}
```

Rethrowing an Exception

- Exception handling is deferred to the caller method.

main() method

```
try {
    putfirstline("/dev/null/NonExistentPath");
} catch (Exception e) {
    System.err.println("Exception at main");
}
```

Output

```
Could not open /dev/null/NonExistentPath
Exception at main
```

Creating Checked Exceptions

- Inherit the Exception classes (subclasses of Throwable) to implement our own exceptions.
  - This will be considered as a checked exception.

```
public class IncorrectFileNameException extends Exception {
  public IncorrectFileNameException(String errorMessage) {
    super(errorMessage);
  }
}
```

Creating Checked Exceptions

- Inherit the Exception classes (subclasses of Throwable) to implement our own exceptions.

```
String filename = "&^%$@";
try {
  if (filename.contains("%") || filename.contains("&")) {
    throw new IncorrectFileNameException("");
  }
  Scanner file = new Scanner(new File(filename));
} catch (IncorrectFileNameException e) {
  System.out.println("Incorrect File name.");
} catch (FileNotFoundException e) {
}
```

Output

Incorrect File name.

Creating Unchecked Exceptions

- Inherit the RuntimeException classes to implement our own unchecked exceptions.
  - This will be considered as an unchecked exception.

```
class CapacityExceedException extends RuntimeException {
    public CapacityExceedException () { }
    public CapacityExceedException(String message) {
        super(message);
    }
}
```

Creating Unchecked Exceptions

- Inherit the RuntimeException classes to implement our own unchecked exceptions.

main() method

```
List<Integer> integerList = new ArrayList<>();
for (int i = 0; i < 999; i++) {
  integerList.add(10);
  if (integerList.size() > 100) {
    throw new CapacityExceedException("");
  }
}
```

Output

```
Exception in thread "main"
CapacityExceedException: at
Test.main(Test.java:10)
```

Throws

- Throws to specify which exception could be thrown during the execution of a method.
  - Written with method signature or class declaration.
  - If a method does not have “throws” statement in its declaration, it means that 1) no checked exception will happen or 2) a checked exception may happen but it will be caught with “try-catch” inside the method.

Throws Example

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

main() method

```
try {
    fun();
} catch (IllegalAccessException e) {
    System.out.println("caught in main.");
}
```

Output

```
Inside fun().
caught in
main.
```

Exceptions in Constructors

```
class InputFile{
  private BufferedReader in;
  InputFile(String fname) throws Exception {
    try { in = new BufferedReader(new FileReader(fname)); }
    catch (FileNotFoundException e) {
      System.err.println("Could not open " + fname);
      throw e;
    }
    catch (Exception e) {
      try { in.close(); }
      catch (IOException e2) {
        System.err.println("in.close() failed");
      }
      throw e;
    }
  }
}
```

Exceptions in Inheritance

- If superclass constructor *throws* a checked exception, the subclass constructor also throws the checked exception (or its superclass exceptions).

```
class Base{
  Base() throws IOException {
    throw new IOException ();
  }
}
class Derived extends Base {
  Derived() throws IOException {
  }
}
```

main() method

```
try{
    Derived d = new Derived();
} catch (IOException e) {
    System.out.println(
      "IOException");
  }
}
```

Output

```
IOException
```

Exceptions in Inheritance

- Overriding method *throws* the same-typed exception (or the subclass exceptions) of the parent method.

```
class Thrower {
    public void print() throws IOException {
        throw new IOException("IO");
    }
}
class MoreThrower extends Thrower {
    @Override
    public void print() throws IOException {
        super.print();
    }
}
```

Exceptions in Inheritance

- Overriding method *throws* the same-typed exception (or the subclass exceptions) of the original method .

main() method

```
Thrower more = new MoreThrower();
try {
  more.print();
} catch (IOException e) {
  System.out.println("IOCaught");
}
```

Output

```
IOCaught
```

Guidelines

- Prefer more specific exceptions
  - Find an exception that **best** suits your exceptional event, and throw it instead of a general *Exception*.
  - E.g. *NumberFormatException* instead of *Exception*
- Catch the most specific exception first
  - If you catch the less specific exception like Exception or IOException in front of FileNotFoundException, it will never reach the block catching more specific ones.
- Exceptions with descriptive messages
  - When throwing an exception, describe the exceptional event as precise as possible for others to understand.

Guidelines

- Don’t catch Throwable
  - It will also catch Errors, serious problems that are not intended to be handled by an application.
- Don’t just log and rethrow
  - Multiple error messages for the same exception, which are just duplicates.
- Don’t ignore exceptions
  - Catching an exception, and doing nothing inside, thinking it will never happen, will cause a problem.
- Refer to http://wiki.c2.com/?ExceptionPatterns for design patterns of exception handling.

Criticism

- Exception handling is often not implemented correctly in software, especially when there are multiple possible sources of exceptions.
  - Unskilled programmers can easily make it faulty.
- Creates hidden and obfuscated control-flow paths that are difficult for programmers to reason about.
- Increase the risk of resource leaks (Files not closed, mutex etc.)

[Weimer, W; Necula, G.C. (2008). "Exceptional Situations and Program Reliability" (PDF). *ACM Transactions on Programming Languages and Systems*. **30** (2). Archived (PDF) from the original on 2015-09-23.]

Criticism. In *Wikipedia*. Retrieved October 22, 2019, from https://en.wikipedia.org/wiki/Exception_handling#Criticism

Summary

- Exception handling deals with errors to provide robust and fault-tolerant programming.
- Exception is handled with try-catch statement, can be thrown by throw keyword, and cleaned up with finally.