Object-Oriented Programming

Exception Handling

CST & DS

United International College

Review

- OOP Design
- `class`
- Keyword: `new`
- Constructors
- Method / constructor overloading
- Keyword: `this`
- Keyword: `static`
- Keyword: `final`
- `package` & `import`

- Modifiers: `private`, `public`, default, `protected`
- Class (object) relationships
- Keyword: `extends`
- Method overriding
- `Object` class: `toString` & `equals`
- Upcasting & downcasting
- Polymorphism
- `abstract` method and class
- `interface` & `implements`

Outline

- Exceptions and errors
- Checked and unchecked exceptions
- **`try`**, **`catch`**, and **`finally`** statements
- The **`throw`** statement
- The **`throws`** clause
- Extending the **`Exception`** class

Programming Errors

- Compile-time errors: lexical / syntactic / semantic.
  - Unknown code: `int 5x = 3;`
  - Missing semicolon: `int x = 3`
  - Undeclared variable: `x = 3;`
- Runtime errors:
  - Error
    - Usually unrecoverable, very rare.
    - Examples: JVM out of memory, hardware error, etc.
  - Exception
    - Often recoverable, fairly common.
    - Examples: file not found (ask the user whether the file should be created; whether the file name is correct), network connection failed (ask the user whether the server name is correct; try again after a short delay), etc.

Introduction to Exceptions

An exception in Java is any abnormal

- unexpected event
- or extraordinary condition

that occurs at runtime.

Examples: file not found exception
array out of bounds exception
unable to get connection exception
etc.

Introduction to Exceptions

```
public class Test1 {
    public static void division(int i, int j) {
        System.out.print(i + " / " + j + " = ");
        int result = i / j;
        System.out.println(result);
    }
    public static void main(String[] args) {
        division(100, 4);
        division(100, 0); // ArithmeticException: / by zero
        System.out.println("End of main()"); // Unreachable
    }
}
```

```
100 / 4 = 25
100 / 0 = Exception in thread "main" java.lang.ArithmeticException:
/ by zero
        at a/Exception.Test.division(Test.java:6)
        at a/Exception.Test.main(Test.java:11)
```

Introduction to Exceptions

Java is safe: every time something wrong happens in your program, Java is guaranteed to:

- Automatically detect that something is wrong.
- Immediately stop the program.
- Show you an exception about the problem.
- Tell you what the exception is about.
- Tell you exactly where the exception happened in your code.

Introduction to Exceptions

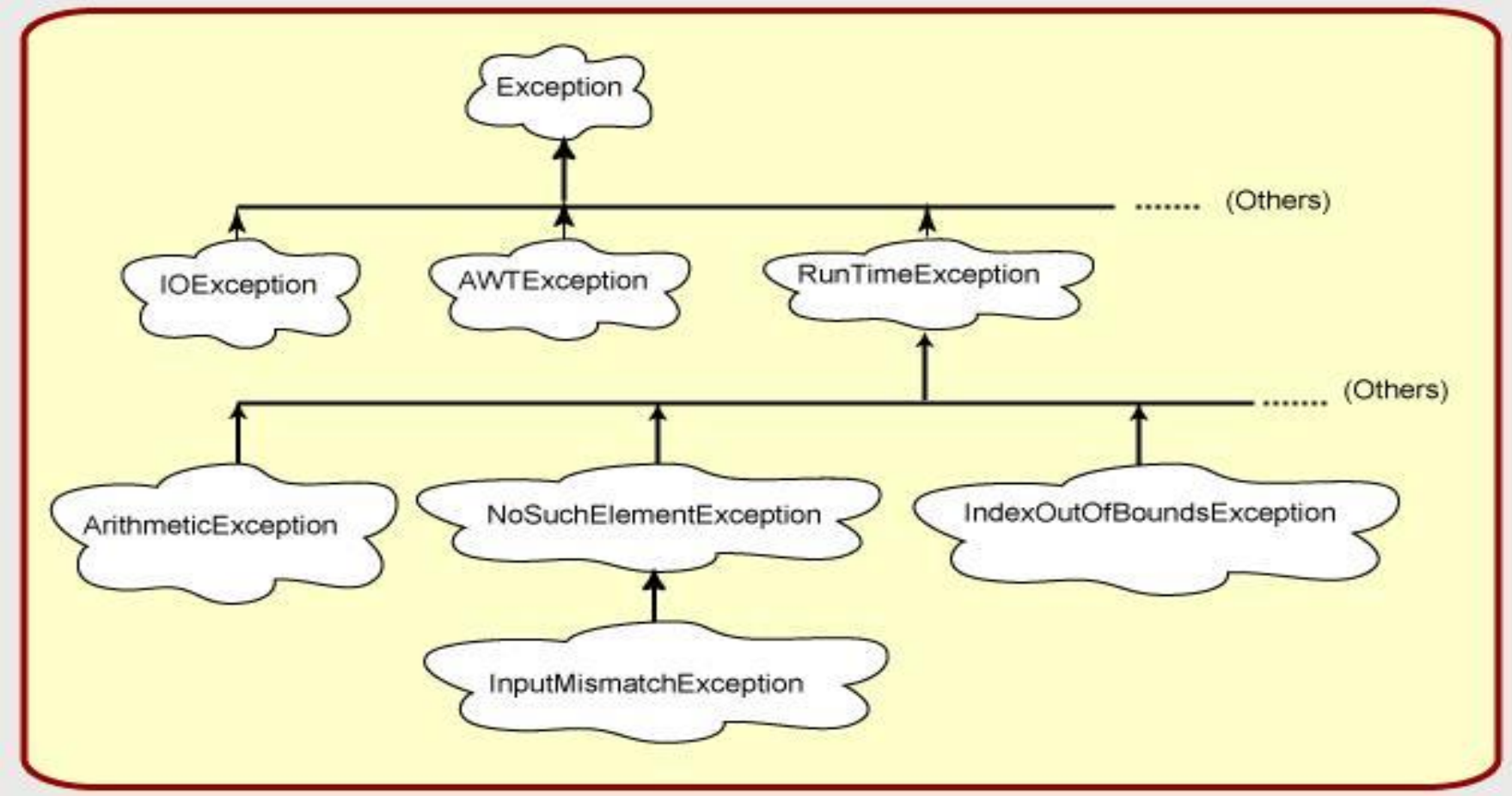
- When an exception occurs, the JVM automatically creates an object from the class **`Exception`** which contains information about the problem. This object represents the exception itself!
- This is called throwing an exception.

- Your Java program may **`catch`** the exception object. Your program can then use the object to recover from the problem.
- This is called handling the exception.

Exception Classes

- The predefined class **`Exception`** is the root class for all possible exceptions
- Every exception class is a descendent class of the class **`Exception`**
- Although the **`Exception`** class can be used directly in a class or program, it is most often used to define a subclass
- The class **`Exception`** is in the **`java.lang`** package

Types of Exceptions



Note: Many exception classes must be imported in order to use them.

E.g: `import java.io.IOException;`

Checked and Unchecked Exceptions

Unchecked exceptions are the class `RuntimeException` and any of its subclasses.

- These exceptions correspond to bugs (divisions by zero, array access out of bounds, etc.) that can happen anywhere in your program.
- The compiler does not force the programmer to handle these exceptions.
- In fact, the programmers may not even know that these exceptions could be thrown.
- Since an unchecked exception corresponds to a bug, it is normal for the program to die (instead of trying to recover from the problem); you must then find the bug in your program and fix it.

Checked and Unchecked Exceptions

Checked exceptions are subclasses of the `Exception` class, excluding the class `RuntimeException` and its subclasses.

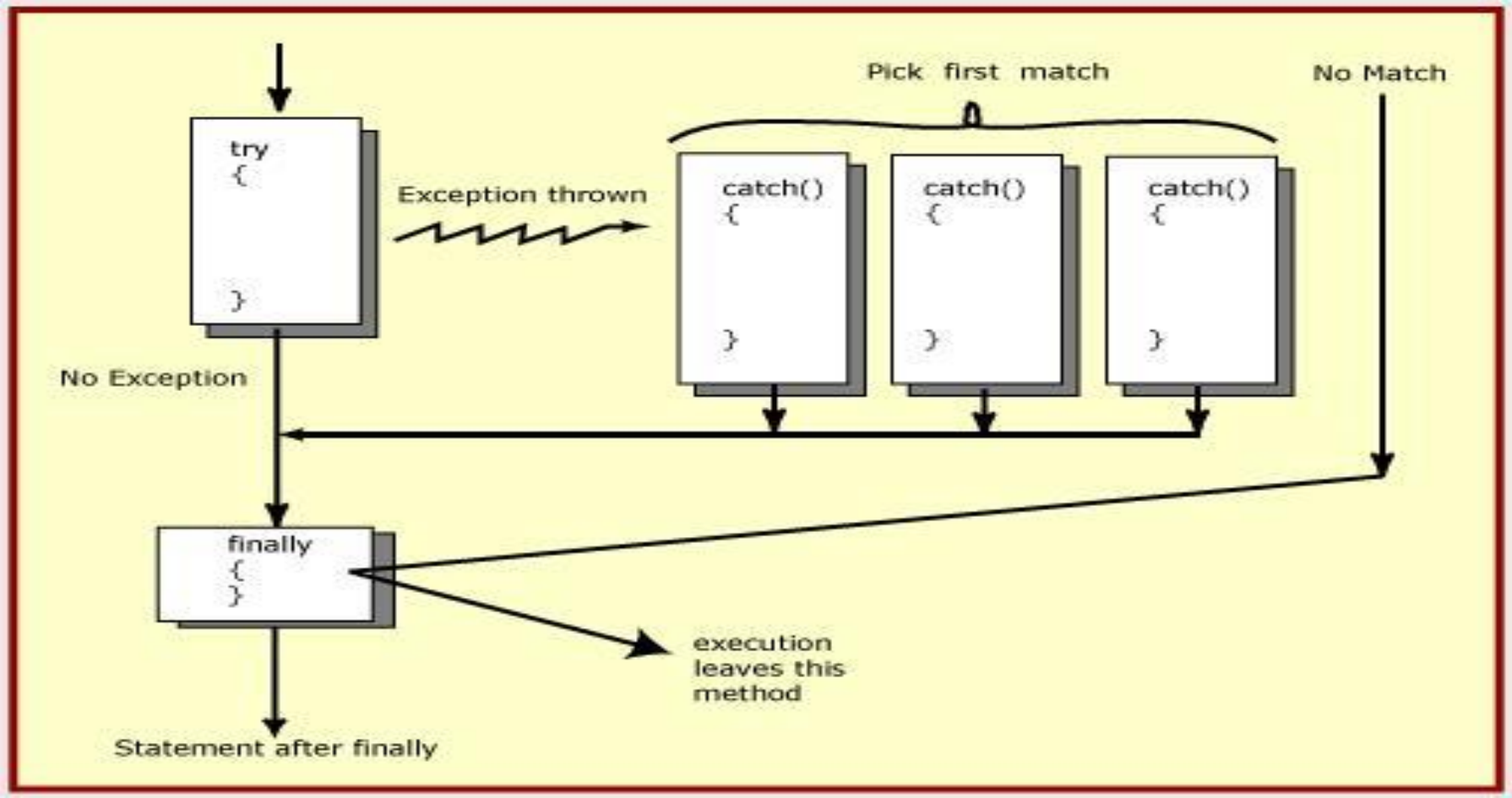
- These exceptions correspond to situation which are unexpected but which are not bugs in the program (the network is temporarily down, the hard disk is full, etc.)
- These exception can happen at specific places in your program (when opening a network connection, when saving a file, etc.)
- The compiler forces the programmer to handle these exceptions in some way (see later).
- Usually the program can recover from the problem (instead of dying) and keep running.

try-catch-finally

Exceptions are handled using a **try-catch-finally** construct, which has the following syntax:

```
try {
        <code that might throw an exception>
} catch (<exception type1> <parameter1>) {
        <statements handling parameter1>
} catch (<exception type2> <parameter2>) {
        <statements handling parameter2>
}
... // as many catch statements as necessary
} finally { // finally block
        <statements always executed>
}
```

try-catch-finally



Note: There can be at most one **`finally`** block, and it must be after all the **`catch`** blocks.

try

- The Java code that you think may produce an exception is placed within a **try** block.
- No exception occurs inside the **try** block → the **finally** block is executed, then the rest of the method.
- An exception occurs inside the **try** block → an exception object is created by Java and thrown, killing all the code after that inside the **try** block, until a matching **catch** block is found and executed to handle the exception object → the **finally** block is executed → the rest of the method is executed.

try

- An exception occurs inside the **try** block → an exception object is created by Java and thrown, killing all the code after that inside the **try** block, and no matching **catch** block is found → the **finally** block is executed → the same exception object keeps killing all the code in the rest of the method → the same exception object keeps killing the rest of the code of the method's caller, and so on, until a matching **catch** is found or the **main** method is killed.

`catch`

- An exception object created and thrown during execution of the **`try`** block can be caught and handled in a **`catch`** block.
- On exit from a **`catch`** block, normal execution continues and the **`finally`** block is executed, followed by the execution of the rest of the method.

finally

- A **finally** block is always executed.
- Generally a **finally** block is used for freeing resources, cleaning up, closing files or network connections etc. Otherwise you do not need it.
- If the **finally** block executes a control transfer statement such as a **return** or a **break** statement, then this control statement determines how the execution will proceed regardless of any **return** or control statement present in the **try** or **catch**.

Example

```
import java.util.Scanner;
public class Test2 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        while(true) {
            System.out.print("Input an integer: ");
            String s = scanner.nextLine();
            int i = Integer.parseInt(s);
            System.out.println("i is: " + i);
        }
    }
}
```

Example

What happens when we run the program and type in a number like **`1.5`**?

```
Exception in thread "main" java.lang.NumberFormatException: For
  input string: "1.5"
  at
  java.base/java.lang.NumberFormatException.forInputString(NumberFor
  matException.java:65)
  at java.base/java.lang.Integer.parseInt(Integer.java:652)
  at java.base/java.lang.Integer.parseInt(Integer.java:770)
  at a/Exception.Test.main(Test.java:8)
```

Example

- The JVM reports that an exception of type `java.lang.NumberFormatException` was **thrown** at line 8 in our code.

- Line 8 is:

```
int i = Integer.parseInt(s);
```

- Let's look at the Java Documentation for the `parseInt` method of the `Integer` class!

Integer.parseInt()

parseInt

```
public static int parseInt(String s)
                    throws NumberFormatException
```

Parses the string argument as a signed decimal integer. The characters in the string must all be decimal digits, except that the first character may be an ASCII minus sign `'-'` (`'\u002D'`) to indicate a negative value or an ASCII plus sign `'+'` (`'\u002B'`) to indicate a positive value. The resulting integer value is returned, exactly as if the argument and the radix 10 were given as arguments to the `parseInt(java.lang.String, int)` method.

Parameters:

`s` - a String containing the int representation to be parsed

Returns:

the integer value represented by the argument in decimal.

Throws:

`NumberFormatException` - if the string does not contain a parsable integer.

Example with try-catch

```
import java.util.Scanner;
public class Test2 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        while(true) {
            System.out.print("Input an integer: ");
            String s = scanner.nextLine();
            try {
                int i = Integer.parseInt(s);
                System.out.println("i is: " + i);
            } catch(NumberFormatException e) {
                System.out.println("Wrong!");
            }
        }
    }
}
```

Example with `try-catch`

Now when we run the program, the exception is caught and handled:

```
Input an integer: 3
i is: 3
Input an integer: 1.5
Wrong!
Input an integer: 4
i is: 4
```

Example with `finally`

```
public class Test3 {
    public static void division(int i, int j) {
        int result = -1;
        try {
            result = i / j;
        } catch(ArithmeticException e) {
            System.out.println("Wrong: " + e.getMessage());
        } finally {
            System.out.println(i + " / " + j + " = " + result);
        }
    }
    public static void main(String[] args) {
        division(100, 4);
        division(100, 0); // ArithmeticException: / by zero
        System.out.println("End of main()");
    }
}
```

Example with `try-catch`

Now when we run the program, the exception is caught and handled:

```
100 / 4 = 25
Wrong: / by zero
100 / 0 = -1
End of main()
```

try-catch-finally

1. For each `try` block there can be zero or more `catch` blocks, but only one `finally` block.
2. The `catch` blocks and `finally` block must always appear in conjunction with a `try` block.
3. A `try` block must be followed by either at least one `catch` block or one `finally` block.
4. The order of the exception handlers in the `catch` blocks must be from the most specific exception to the least specific one.

Example with compile-time error

```
public class Test4 {
    public static void division(int i, int j) {
        int result = -1;
        try {
            result = i / j;
        } catch(Exception e) { // Catches all exceptions!
            System.out.println("Wrong: " + e.getMessage());
        } catch(ArithmeticException e) { // Unreachable code.
            System.out.println("Wrong: " + e.getMessage());
        } finally {
            System.out.println(i + " / " + j + " = " + result);
        }
    }
    public static void main(String[] args) {
        division(100, 4);
        division(100, 0); // ArithmeticException: / by zero
        System.out.println("End of main()");
    }
}
```

Unreachable catch block for ArithmeticException.
It is already handled by the catch block for Exception

Exception information

- `getMessage()`
  - Returns the detailed error message string for this exception object.
- `printStackTrace()`
  - Prints this exception and its backtrace: the list of method calls starting from `main` up to the method that threw the exception object.

Example with `finally`

```
public class Test5 {
    public static void division(int i, int j) {
        int result = -1;
        try {
            result = i / j;
        } catch(ArithmeticException e) {
            System.out.println("Wrong: " + e.getMessage());
            e.printStackTrace();
        } finally {
            System.out.println(i + " / " + j + " = " + result);
        }
    }
    public static void main(String[] args) {
        division(100, 4);
        division(100, 0); // ArithmeticException: / by zero
        System.out.println("End of main()");
    }
}
```

Example with `try-catch`

Now when we run the program, the exception is caught and handled:

```
100 / 4 = 25
Wrong: / by zero
java.lang.ArithmeticException: / by zero
    at a/Exception.Test.division(Test.java:8)
    at a/Exception.Test.main(Test.java:18)
100 / 0 = -1
End of main()
```

`throw` & `throws`

A method can explicitly create an object from the `Exception` class (or one of its subclasses) using `new` and then throw the object using the `throw` statement. Then the method must either:

- Catch the exception object itself using a `try-catch` statement.
  - This is not very useful: in general there is little reason for a method to throw an exception just to immediately catch it in the same method…
- Or use the `throws` clause to specify as part of the type of the method that the method might throw an exception.
  - The exception object will then have to be caught somewhere else, in one of the callers of the method.

throw & throws

```
import java.util.Scanner;
public class Test6 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        while(true) {
            System.out.print("Input a positive integer: ");
            String s = scanner.nextLine();
            try {
                int i = Integer.parseInt(s);
                if(i <= 0) {
                    // This is overkill.  It is easier to just print the
                    // error message here and then use "continue" to start
                    // the next iteration of the while loop.
                    throw new Exception(i + " <= 0 !");
                }
                System.out.println("i is: " + i);
            } catch(Exception e) { // Catch all exceptions!
                System.out.println("Catch: " + e.getMessage());
            }
        }
    }
}
```

throw & throws

```
import java.util.Scanner;
public class Test7 {
    public static void readPosInt() throws Exception {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Input a positive integer: ");
        String s = scanner.nextLine();
        int i = Integer.parseInt(s);
        if(i <= 0) {
            throw new Exception(i + " <= 0 !");
        }
        System.out.println("i is: " + i);
    }
    public static void main(String[] args) {
        while(true) {
            try {
                readPosInt(); // May throw an exception, which we need to catch.
            } catch(Exception e) { // Catch all exceptions!
                System.out.println("Catch: " + e.getMessage());
            }
        }
    }
}
```

throw & throws

Now when we run the program, the exceptions are caught and handled by corresponding handlers:

```
Input a positive integer: -2
Catch: -2 <= 0 !
Input a positive integer: 1.5
Catch: For input string: "1.5"
```

throws

- In a method, if you throw an exception object and you don't want to catch the exception object in the same method then you must use the **throws** clause.

- The method might throw multiple different kinds of exceptions: you need to list in the **throws** clause all the exceptions that the method does not catch itself.

```
public Type methodName(…) throws <Exception1>,…, <ExceptionN> {
        …
}
```

What happens if an exception is never caught?

- In a non-GUI program, the exception object will kill everything (including `main`), the JVM will then catch the exception, and show it to you on the screen.
- In a GUI program, multiple threads are running, and the exception object will only kill the current thread (the one throwing the exception). The other threads will still run but the software as a whole will probably stop working properly.
- So every well-written program should eventually catch every exception by a `catch` block in some method!

Custom new exceptions

- We can have our own custom exception objects to deal with special exception conditions instead of using the existing exception classes.
- Custom exception objects should be created from your own exception classes which are subclasses of Java's **`Exception`** class.

Custom new exceptions

1. Define a subclass of **`Exception`**.
2. Create an exception object from the subclass using **`new`**.
3. Throw the exception object using **`throw`**.
4. Use a **`try-catch`** statement to handle the exception object inside the method itself, or use a **`throws`** clause and catch the exception object somewhere else.

Example

```
public class NotPositiveException extends Exception {
    public NotPositiveException(String msg) {
        // msg is the message given to the exception
when
        // it is created, which will later be the
result
        // of calling the getMessage() method.
        super(msg);
    }
}
```

Example

```
import java.util.Scanner;
public class Test8 {
    public static void readPosInt() throws NotPositiveException
{
        Scanner scanner = new Scanner(System.in);
        System.out.print("Input a positive integer: ");
        String s = scanner.nextLine();
        int i = Integer.parseInt(s);
        if(i <= 0) {
            throw new NotPositiveException(i + " <= 0 !");
        }
        System.out.println("i is: " + i);
    }
    ...
```

Example

```
    ...
    public static void main(String[] args) {
        while(true) {
            try {
                readPosInt();
            } catch(NotPositiveException e) { // If i <= 0.
                System.out.println("NotPositive: " + e.getMessage());
            } catch(NumberFormatException e) { // If parseInt() fails.
                System.out.println("NumberFormat: " + e.getMessage());
            }
        }
    }
}
```

```
Input a positive integer: -2
NotPositive: -2 <= 0 !
Input a positive integer: -2.5
NumberFormat: For input string: "-2.5"
```

`throws` and method overidding

- An overriding method in subclass can throw any unchecked exception, regardless of whether the overridden method in superclass throws exceptions or not.
- If an overriding method in subclass throws checked exception, then it must be the same exception or child exceptions of the exception(s) declared in the overridden method.

throws and inheritance

```
class A{
        void meth() throws FileNotFoundException {}
}
class B extends A {
        @Override
        void meth () {} // compiles fine
}
class C extends A {
        @Override
        void meth () throws  IOException {} // compile error
        /* Exception IOException is not compatible with throws clause
         * in A.meth()  */
}
class D extends A
{
        @Override
        void meth() throws RuntimeException{}// compiles fine
}
```

`throws` and inheritance

```
interface A {
        void meth() throws IOException;
}
class B implements A {
        @Override
        public void meth() throws FileNotFoundException { }
        // compiles fine
}
class C implements A  {
        @Override
        public void meth() { } // compiles fine
}

class D implements A  {
        @Override
        public void meth() throws Exception { } // compile error
}
```

Summary

- Exceptions and errors
- Checked and unchecked exceptions
- `try`, `catch`, and `finally` statements
- The `throw` statement
- The `throws` clause
- Extending the `Exception` class