JAVA

Lecture V – Exception Handling

RECAP

- Basic OOP
- Classes definition
- Variables declaration
- Methods definition
- Constructor
- this
- static vs non-static
- Encapsulation
- Call by reference vs call by value

EXCEPTIONS

```
int n = “Integer”;

int[] xs = new int[3];
xs[7] = 7;
```

What are the problems?

EXCEPTIONS

```
int n = "Integer";        // complier-time error

int[] xs = new int[3];
xs[7] = 7;
```

What are the problems?

EXCEPTIONS

```
int n = "Integer";          // complier-time error

int[] xs = new int[3];
xs[7] = 7;                  // it compiles, but fails when JVM
                            // tries to execute it
```

What are the problems?

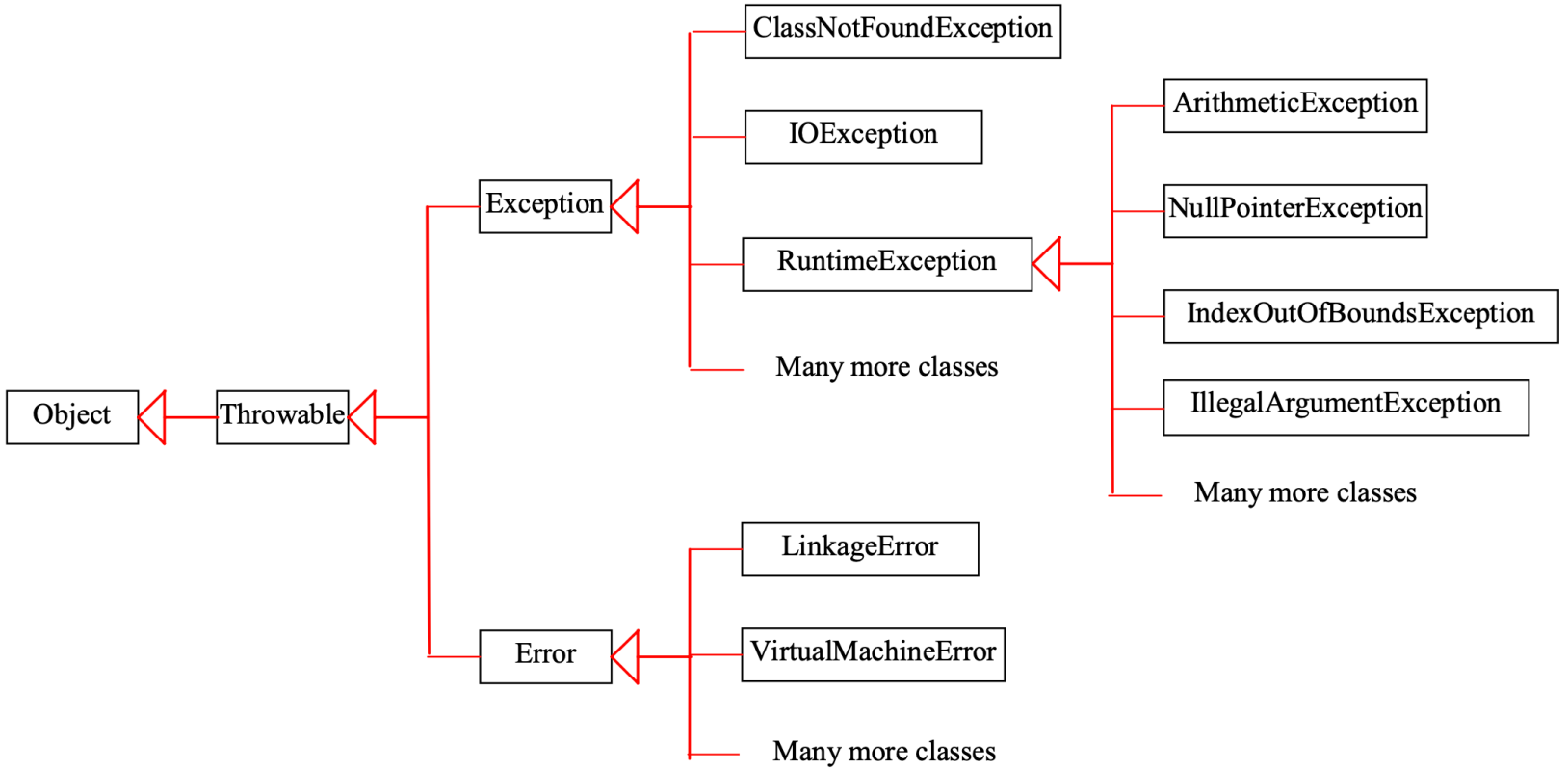
EXCEPTIONS

- Compile-time error:
  - syntax errors, int n = “Integer”;
  - Detected by the Java compiler.
  - Cannot be executed
- Run-time error: exceptions only happens at run time.
  - Index out of range… most programming logic problems
  - Detected by the JVM.
  - Can be executed.
- Run-time error can be handled by a block of code called exception handler.

EXCEPTION HIERARCHY

- All runtime errors are classes derived from a class called **Throwable**, i.e., java.lang.Throwable
- Two direct subclass of Throwable:
  - Error: System Errors beyond your control, e.g., those in JVM. There is little you can do…
  - Exception: Errors that result from program activity.
    - RuntimeException: caused by programming errors
- We focus on handling exceptions…

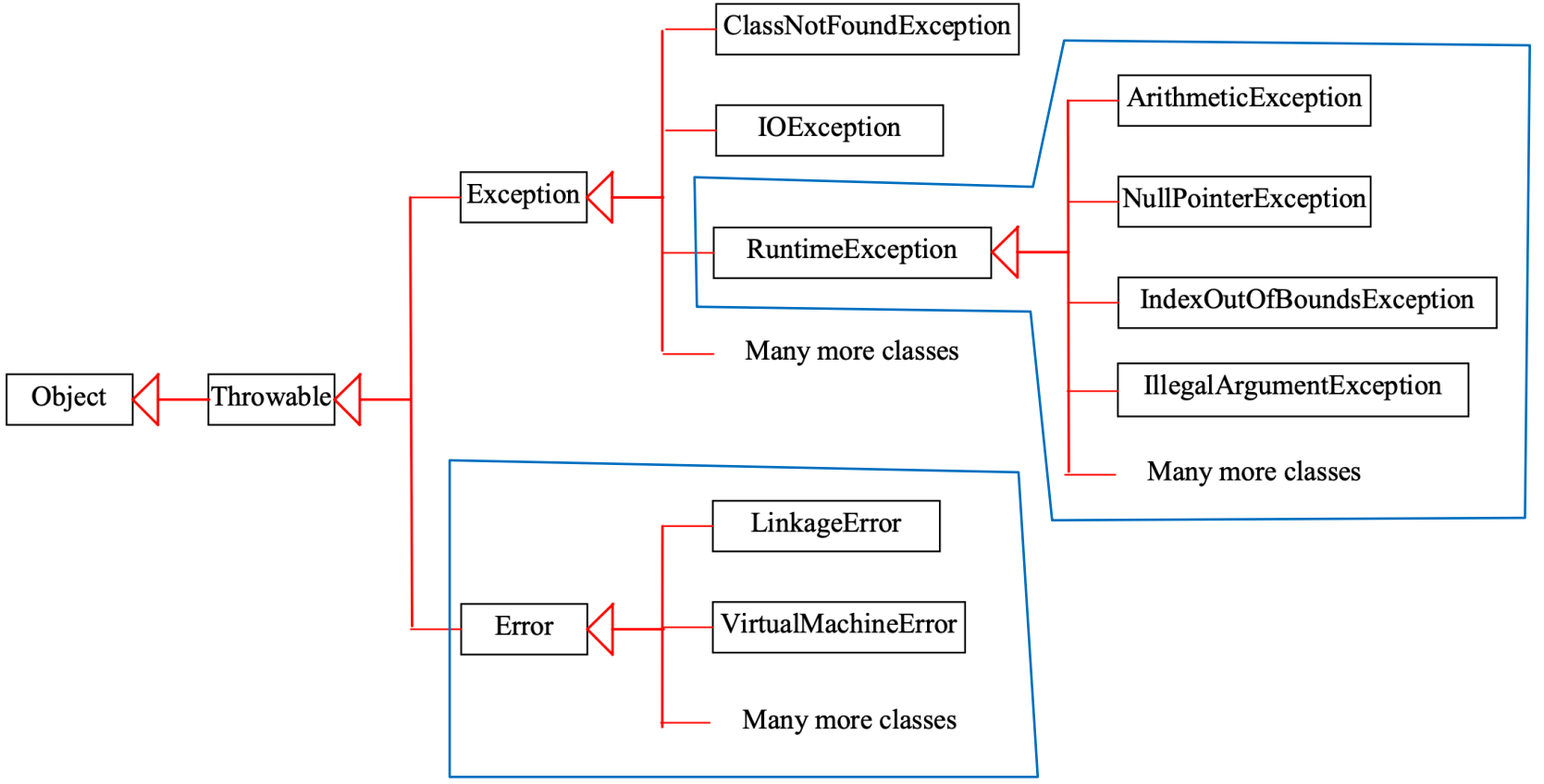
EXCEPTION HIERARCHY



CHECKED VS UNCHECKED EXCEPTION

- Unchecked Exceptions: RuntimeException, Error and their subclasses.
  - Reflect programming logic errors that are not recoverable
  - Can occur anywhere in the program
  - Java does not force you to catch.
- Checked Exceptions: All other exceptions are known as checked exceptions. The compiler forces the programmer to check and deal with such exceptions, e.g., IOException

CHECKED VS UNCHECKED EXCEPTION



UNCHECKED EXCEPTIONS

Exception	Meaning
ArithmeticException	Arithmetic errors
NullPointerException	Invalid use of a null reference
NumberFormatException	Invalid conversion of a string to a numeric format
ClassCastException	Invalid cast
ArrayIndexOutOfBounds	Array index is out-of-bounds
IndexOutOfBounds	Some type of index is out of bouds
StringIndexOutOfBouds	Attempt to index outside the bounds of a string
…	…

EXCEPTION HANDLING

- Five keywords in exception handling:
  - **try**: used to monitor exceptions, i.e., exception occurs within the try block will be thrown
  - **catch**: used to catch the exceptions and define how to handle it.
  - **throw**: used to manually throw an exception.
  - **throws:** used in the method signature to declare an exception which might be thrown by executing this method.
  - **finally**: to declare any code that must be executed upon exiting from a try block.

TRY AND CATCH

- **try** and **catch** work together which is the core of exception handling, i.e., you cannot have a catch without try.

```
try{
        // block of code to monitor
}catch(ExceptionType1 exOb){
        // handler for exception 1
}catch(ExceptionType2 exOb){
        // handler for exception 2
}
```

...

- The exception happens within try block will be caught by corresponding catch and then handled.

EXAMPLE

```
class ExcDemo1{
      public static void main(String[] args){
            int[] nums = new int[4];
            try{  // create a try block
                  System.out.println("Before");
                  nums[7] = 10;
                  System.out.println("After");
            }catch(ArrayIndexOutOfBoundsException e){
                  System.out.println("Index out of bounds!");
            }
            System.out.println("Catch finishes");
      }
}
```

EXAMPLE2

- Exceptions in Method call can also be caught.

```
class ExcDemo2{
	public static void genException(){
		int[] nums = new int[4];
		nums[7] = 0;
	}
	public static void main(String[] args){
		try{  // create a try block
			genException();
		}catch(ArrayIndexOutOfBoundsException e){
			System.out.println("Index out of bounds!");
		}
		System.out.println("Catch finishes");
	}
}
```

EXCEPTION HANDLING

- Catching an unchecked exception:
  - Prevents abnormal program termination. (program will terminate if the execution is handled)
  - Exception handling is useful, e.g., you want re-prompt the user to enter a valid input.
  - Exception is not fixed, e.g., a network connection might time-out
  - When we don't know what exception will happen, we could refer it to Excepetion e and then find out what e by calling e.printStackTrace();

EXAMPLE 3

- You are given an array of integers (which may contain 0)
- You need to calculate the value of 80 divided by each integer in this given array and then add them together.
- If there is a division by 0, report the error and continue.

EXAMPLE 3

- Catching the ArithmeticException and continue:

```
class ExcDemo3{
	public static void main(String[] args){
		int[] nums = {4, 0, 8, 16};
		int total = 0;
		for(int n : nums){
			try{
				System.out.println(80 / n);
				total += 80/n;
			}catch(ArithmeticException e){
				System.out.println("Can't divide by Zero");
			}
		}
	}
}
```

MULTIPLE CATCH BLOCKS

- It is possible to catch multiple exceptions by using more than one catch clauses.
- But each **catch** must catch a different type of exception.

```
int[] nums = {4, 2, 8, 16};
int[] nums2 = {0, 3};
for(int i = 0; i < nums.length; i++){
      try{
            System.out.println(nums[i] / nums2[i]);
      }catch(ArithmeticException e){
            System.out.println("Can't divide by Zero");
      }catch(ArrayIndexOutOfBoundsException e){
            System.out.println("Index out of bounds");
      }
}
```

NESTED TRY BLOCKS

```
static void method2(){
  try{
        // invoke method 3
        method3();
        System.out.println("After Method
3");
  }catch(ArithmeticException e){
        System.out.println("Exception
                        Method 2");
  }
}
```

```
static void method3(){
  try{
        int x = 4 / 0;
        System.out.println(”ERROR Here");
  }catch(ArithmeticException e){
        System.out.println("Exception
                        Method 3");
  }
}
```

```
static void method1(){
  try{
        // invoke method 2
        method2();
        System.out.println("After Method
2");
  }catch(ArithmeticException e){
        System.out.println("Exception
                        Method 1");
  }
}
```

```
public static void main(String[] args){
  try{
        method1();
        System.out.println("After Method
1");
  }catch(ArithmeticException e){
        System.out.println("Exception in
main");
  }
}
```

THROWING AN EXCEPTION

- In preceding examples, exceptions are caught automatically by JVM.
- It is also possible to manually throw an exception by using **throw** keyword.
- `throw exceptOb;`
- `exceptOb` is an object of an exception class

```
try{
      System.out.println("Before");
      throw new ArithmeticException();
}catch(ArithmeticException exc){
      System.out.println("Exception caught");
}
```

RETHROWING AN EXCEPTION

- An exception could be rethrown so that it can be caught by an outer catch

```
try{
      try{
            System.out.println(“Before”);
            throw new ArithmeticException();
      }catch(ArithmeticException exc){
            System.out.println(“Exception caught”);
            throw new ArithmeticException();
      }
}catch (ArithmeticException exc){
      System.out.println(”outer”);
}
```

FINALLY

- What will happen here?

```
try{
        System.out.println("Before");
        throw new ArithmeticException();
}catch(ArithmeticException exc){
        System.out.println("Exception caught");
        return;
}
System.out.println("After");
```

FINALLY

- We may want to execute some code when a try/catch block is left (No matter there is an exception or not).
- For example, we may have allocated some resources that needs to be released.

```
try{
        System.out.println("Before");
        throw new ArithmeticException();
}catch(ArithmeticException exc){
        System.out.println("Exception caught");
        return;
}
System.out.println("After");  // it won't work here
```

TRACE A PROGRAM I

```
try{
        statements;
}catch(ExceptionType ex){
        handling ex;
}finally{
        finalStatement;
}
nextStatement;
```

If no exception happens

TRACE A PROGRAM I

```
try{
        statements;
}catch(ExceptionType ex){
        handling ex;
}finally{
        finalStatement;
}
nextStatement;
```

The final block is always executed

TRACE A PROGRAM I

```
try{
        statements;
}catch(ExceptionType ex){
        handling ex;
}finally{
        finalStatement;
}
nextStatement;
```

Then the next statement will be executed

TRACE A PROGRAM II

```
try{
        statement1;
        statement2;
}catch(ExceptionType ex){
        handling ex;
}finally{
        finalStatement;
}
nextStatement;
```

If there is an exception of type ExceptionType happens

TRACE A PROGRAM II

```
try{
        statement1;
        statement2;
}catch(ExceptionType ex){
        handling ex;
}finally{
        finalStatement;
}
nextStatement;
```

The exception is handled

TRACE A PROGRAM II

```
try{
        statement1;
        statement2;
}catch(ExceptionType ex){
        handling ex;
}finally{
        finalStatement;
}
nextStatement;
```

The final block is always executed

TRACE A PROGRAM II

```
try{
        statement1;
        statement2;
}catch(ExceptionType ex){
        handling ex;
}finally{
        finalStatement;
}
nextStatement;
```

The next statement is then executed

TRACE A PROGRAM III

```
try{
        statement1;
        statement2;
}catch(ExceptionType ex){
        handling ex;
        throw ex;    // rethrow
}finally{
        finalStatement;
}
nextStatement;
```

If there is an exception of type ExceptionType happens

TRACE A PROGRAM III

```
try{
        statement1;
        statement2;
}catch(ExceptionType ex){
        handling ex;
        throw ex;    // rethrow
}finally{
        finalStatement;
}
nextStatement;
```

The exception is handled

TRACE A PROGRAM III

```
try{
        statement1;
        statement2;
}catch(ExceptionType ex){
        handling ex;
        throw ex;    // rethrow
}finally{
        finalStatement;
}
nextStatement;
```

Execute the final block

TRACE A PROGRAM III

```
try{
        statement1;
        statement2;
}catch(ExceptionType ex){
        handling ex;
        throw ex;    // rethrow
}finally{
        finalStatement;
}
nextStatement;
```

Rethrow the exception and the control is transferred to the caller

THROWS

- If a method generates an exception that it does not handle, then it must declare that exception

```
type name(param-list) throws except-list{
        // body
}
```

- `except-list`: a list of exceptions the method might throw outside of itself.
- Why we didn't need to specify a throws clauses in our previous examples?

THROWS

- If a method generates an exception that it does not handle, then it must declare that exception

```
type name(param-list) throws except-list{
        // body
}
```

- `except-list`: a list of exceptions the method might throw outside of itself.
- Why we didn't need to specify a throws clauses in our previous examples?
  - Unchecked Exceptions vs Checked Exceptions
  - All checked exceptions need to be declared. (what happens if not?)

- Example: IOException (more about I/O later)

THROWS

```
public static char prompt(String str){
      System.out.println(str + ": ");
      return (char) System.in.read(); // a method to read the next
                                          byte from the input
}
public static void main(String[] args){
      char ch = 'X';
      ch = prompt("Enter a letter");
      System.out.println("You pressed " + ch);
}
```

THROWS

```
public static char prompt(String str) throws java.io.IOException{
      System.out.println(str + ": ");
      return (char) System.in.read(); // a method to read the next
                                          byte from the input
}
public static void main(String[] args){
      char ch = 'X';
      try{
            ch = prompt("Enter a letter");
      }catch(java.io.IOException exc){
            System.out.println("I/O exception occurred.");
      }
      System.out.println("You pressed " + ch);
}
```

THROWS VS EXCEPTION HANDLING

- When to use **throws**:
  - All checked exceptions must be thrown.
  - If you want the exception to be processed by its caller.
- When to use **try-catch** block:
  - To deal with unexpected error conditions.
  - If you want to handle the exception in the method where it occurs.

MULTI-CATCH

- New feature from JDK 7.
- Two or more exceptions can be caught by the same **catch** clause.
- Previously, we use catch for each potential exception.
  - Potentially code duplication
- General form:

```
catch( ExceptionType1 | ExceptionType2 e)
```

- Each parameter is implicitly **final**

```
catch( final ExceptionType1 | ExceptionType2 e)
```