Object
Data Abstraction
Class
OOps concept in Java
Encapsulation
Inheritance
Polymorphism

CS20004: Object Oriented Programming using Java

Lec-19

In this Discussion . . .

Exceptions

- Exception is an abnormal condition that arises when executing a program. *Exception is an error which can be handled. But, an error is an error which can’t be handled*
- In the languages that do not support exception handling, errors must be checked and handled manually, usually through the use of error codes

Exceptions

- In the languages that do not support exception handling, errors must be checked and handled manually, usually through the use of error codes
- **In contrast, Java:**
  - provides syntactic mechanisms to signal, detect and handle errors
  - ensures a clean separation between the code executed in the absence of errors and the code to handle various kinds of errors
  - brings run-time error management into object-oriented programming

Exception Handling

- An **exception** is an object that describes an exceptional condition (error) that has occurred when executing a program
- **Exception handling involves the following:**
  - when an error occurs, an object (exception) representing this error is created and thrown in the method that caused it
  - that method may choose to handle the exception itself or pass it on
  - either way, at some point, the exception is caught and processed

Exception Sources

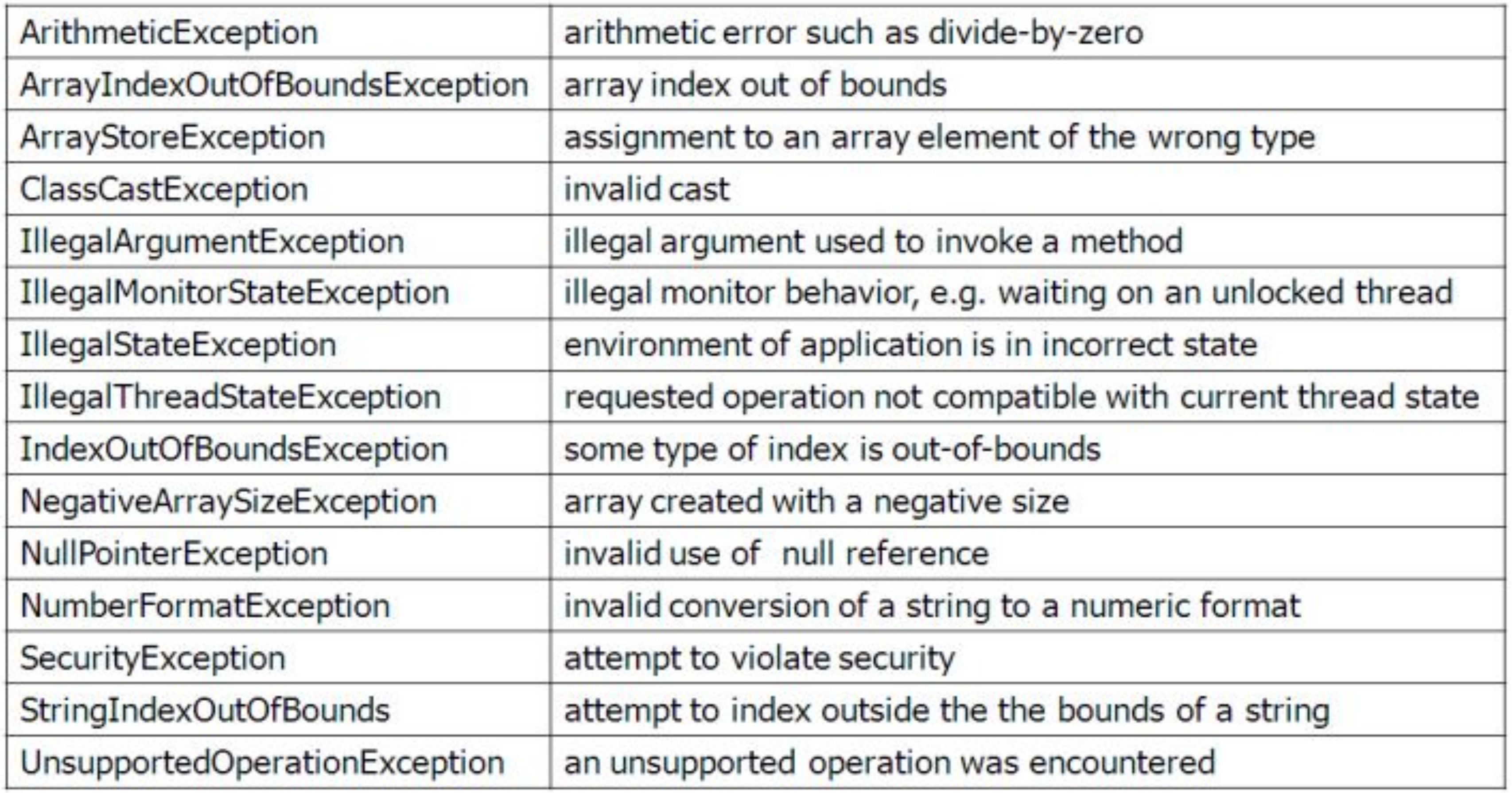
- **Exceptions can be:**
  - generated by the Java run-time system
    - Fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment
  - manually generated by programmer’s code
    - Such exceptions are typically used to report some error conditions to the caller of a method

Java Built-In Exceptions

- The default java.lang package provides several exception classes, all sub-classing the RuntimeException class
- **Two sets of build-in exception classes:**
  - unchecked exceptions
    - the compiler does not check if a method handles or throws there exceptions
  - checked exceptions
    - must be included in the method’s throws clause if the method generates but does not handle them

Unchecked Built-In Exceptions

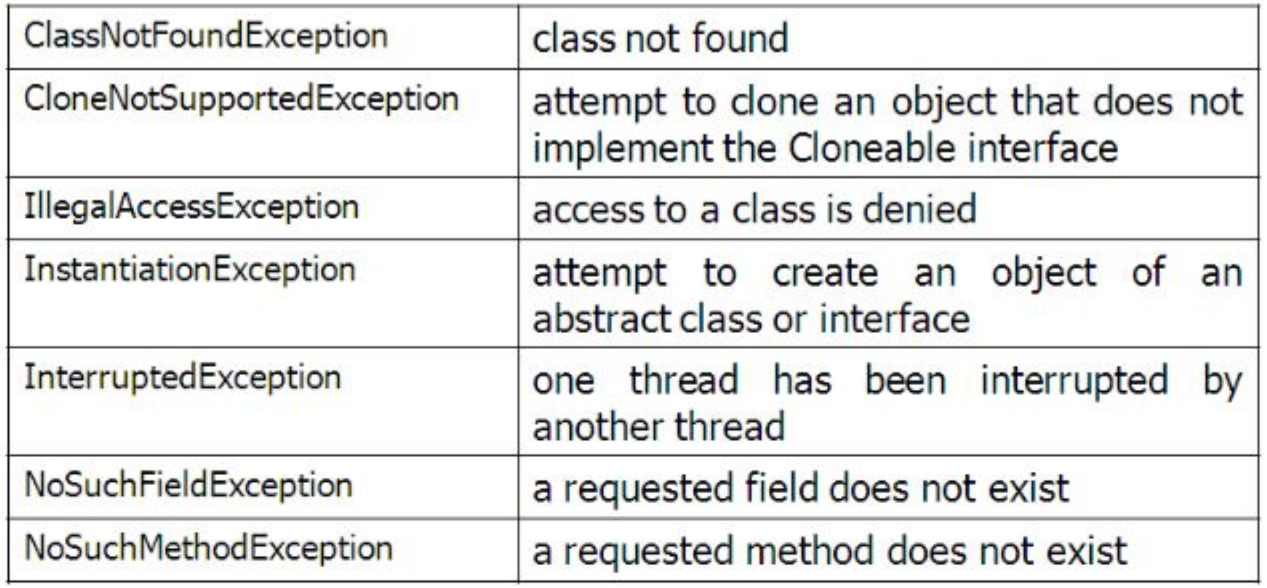
- These are the exceptions that are not checked at compile time and checked by the JVM



ArithmeticException	arithmetic error such as divide-by-zero
ArrayIndexOutOfBoundsException	array index out of bounds
ArrayStoreException	assignment to an array element of the wrong type
ClassCastException	invalid cast
IllegalArgumentException	illegal argument used to invoke a method
IllegalMonitorStateException	illegal monitor behavior, e.g. waiting on an unlocked thread
IllegalStateException	environment of application is in incorrect state
IllegalThreadStateException	requested operation not compatible with current thread state
IndexOutOfBoundsException	some type of index is out-of-bounds
NegativeArraySizeException	array created with a negative size
NullPointerException	invalid use of null reference
NumberFormatException	invalid conversion of a string to a numeric format
SecurityException	attempt to violate security
StringIndexOutOfBounds	attempt to index outside the the bounds of a string
UnsupportedOperationException	an unsupported operation was encountered

Checked Built-In Exceptions

- These are the exceptions that are checked at compile time by the java compiler.
- If some code within a method throws a checked exception,then the method must either handle the exception or it must specify the exception using *throws* keyword



ClassNotFoundException	class not found
CloneNotSupportedException	attempt to clone an object that does not implement the Cloneable interface
IllegalAccessException	access to a class is denied
InstantiationException	attempt to create an object of an abstract class or interface
InterruptedException	one thread has been interrupted by another thread
NoSuchFieldException	a requested field does not exist
NoSuchMethodException	a requested method does not exist

Exception Constructs

- Five constructs are used in exception handling:

try	a block surrounding program statements to monitor for exceptions
catch	together with try, catches specific kinds of exceptions and handles them in some way
finally	specifies any code that absolutely must be executed whether or not an exception occurs

Exception Constructs

- Five constructs are used in exception handling:

throw	used to throw a specific exception from the program
throws	specifies which exceptions a given method can throw

Exception Handling Block

```
try
{
        //Statements
}
catch(Exception1 ex1)
{
        //Statements
}
catch(Exception2 ex2)
{
       //Statements
}
finally
{
      //Statements
}
```

- try {...} is the block of code to monitor for exceptions
- catch(Exception ex) {...} is exception handler for the exception Exception
- finally {...} is the block of code that is always executed whether an exception is handled or not.
  - Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.
  - The finally block follows the try-catch block

Exception Hierarchy

- All exceptions are subclasses of the build-in class Throwable
- Throwable contains two immediate sub-classes:

Exception	Error
• refers to the exceptional conditions that programs should catch.	• exceptions used by Java to indicate errors with the run-time environment
• The class includes:	• user programs are not supposed to catch them

Runtime Exception	**User-defined Exception Classes**
defined automatically for user programs to include: division by zero, invalid array indexing, etc	

Uncaught Exception

```
class Exc
{
    public static void main(String args[])
    {
        int d = 0;
        int a = 42/d;
        System.out.println(a);
    }
}
```

- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and throws this object.
- This will cause the execution of Exc to stop once an exception has been thrown it must be caught by an exception handler and dealt with.

Default Exception Handler

- As we have not provided any exception handler, the exception is caught by the default handler provided by the Java run-time system

This default handler:

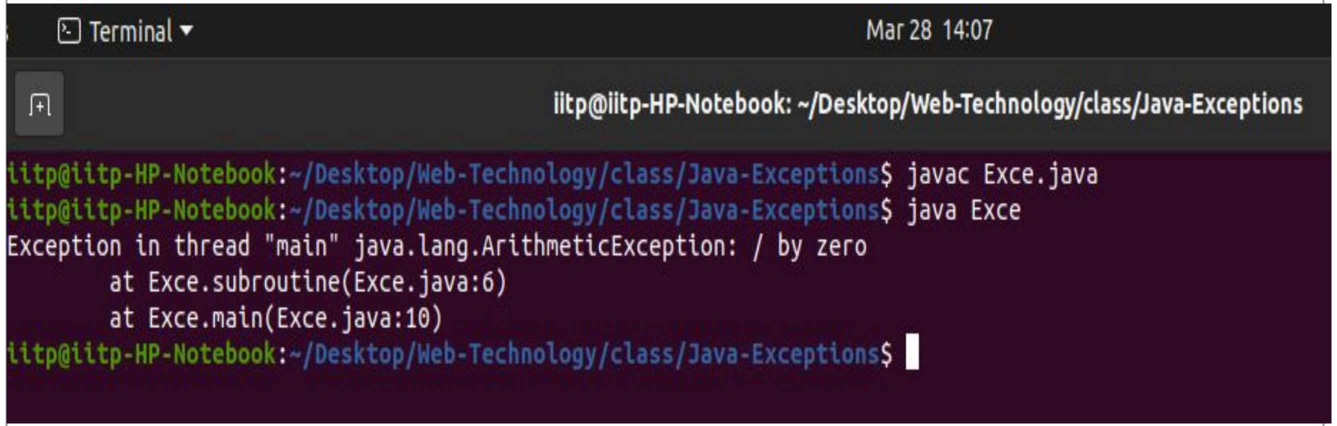
- displays a string describing the exception
- prints the stack trace from the point where the exception occurred
- terminates the program
    - Exception in thread "main" java.lang.ArithmeticException: / by zero at Exc.main(Exc.java:6)

- ***Any exception not caught by the user program is ultimately processed by the default handler***

Stack Trace Display

- Stack trace is actually a record of the active stack frames generated by the execution of a program. It is used for debugging.
- The stack trace displayed by the default error handler shows the sequence of method invocations that led up to the error.
- Here, the exception is raised in subroutine() which is called by main():

```
class Exce
{
        static void subroutine()
        {
                int d = 0;
                int a = 10/d;
        }
        public static void main(String args[])
        {
                Exce.subroutine();
        }
}
```


Terminal
Mar 28 14:07
iitp@iitp-HP-Notebook: ~/Desktop/Web-Technology/class/Java-Exceptions

```
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Exceptions$ javac Exce.java
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Exceptions$ java Exce
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Exce.subroutine(Exce.java:6)
        at Exce.main(Exce.java:10)
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Exceptions$
```

Own Exception Handling

- Default exception handling is basically useful for debugging
- Normally, we want to handle exceptions ourselves because:
  - if we detected the error, we can try to fix it
  - we prevent the program from automatically terminating
- Exception handling is done through the try and catch block

Try and Catch

- try surrounds any code we want to monitor for exceptions
- catch specifies which exception we want to handle and how

```
try
{
    d = 0;
    a = 42/d;
    System.out.println("This will not be printed");
}
```

- control moves immediately to the catch block:

```
catch(Exception e)
{
    System.out.println("Division by Zero");
}
```

The exception is handled and the execution resumes

Try and Catch (Contd.)

- The scope of catch is restricted to the immediately preceding try statement, i.e., it cannot catch exceptions thrown by another try statements
- Resumption occurs with the next statement after the try/catch block:

```
try
{

}
catch(Exception e)
{

}
System.out.println("After Catch Statement");
```

The purpose of catch should be to resolve the exception and then continue as if the error had never happened

Try and Catch (Contd.)

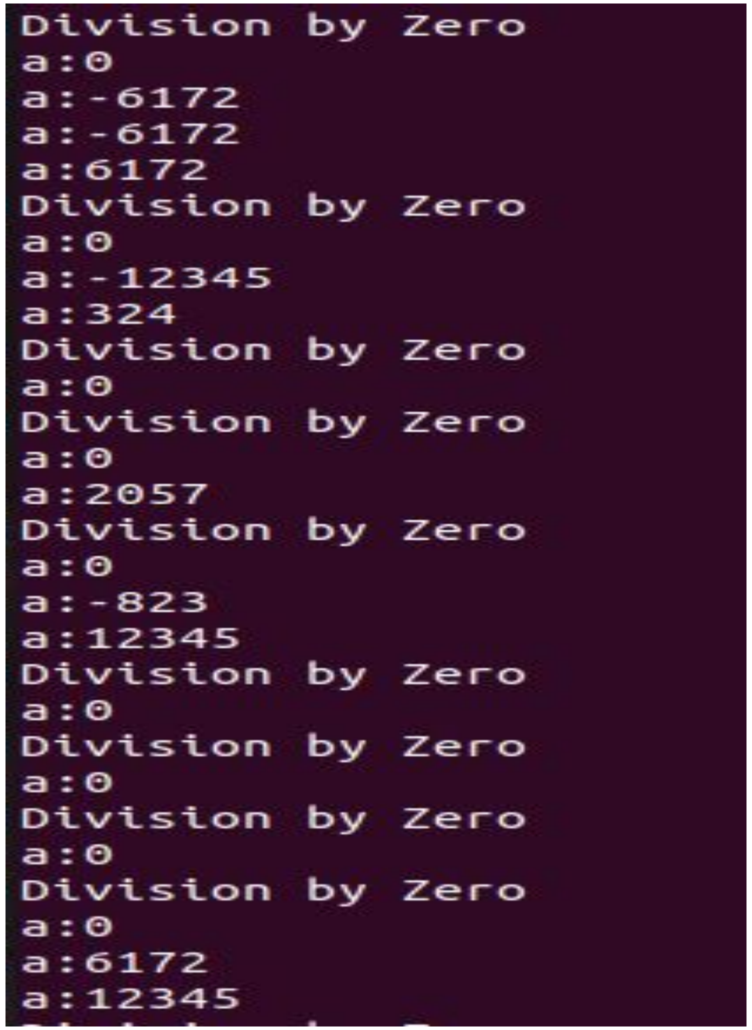
```
import java.util.Random;

class Errorhandle
{
        public static void main(String args[])
        {
                int a=0, b=0,c=0;
                Random r = new Random();
                for(int i =0; i<32000; i++)
                {
                        try
                        {
                                b = r.nextInt();
                                c = r.nextInt();
                                a = 12345/(b/c);
                        }
                        catch(ArithmeticException e)
                        {
                                System.out.println("Division by Zero");
                                a = 0; //set a to zero and continue
                        }
                        System.out.println("a:"+a)
                }
        }
}
```

Try and Catch (Contd.)

```
import java.util.Random;

class Errorhandle
{
	public static void main(String args[])
	{
		int a=0, b=0,c=0;
		Random r = new Random();
		for(int i =0; i<32000; i++)
		{
			try
			{
				b = r.nextInt();
				c = r.nextInt();
				a = 12345/(b/c);
			}
			catch(ArithmeticException e)
			{
				System.out.println("Division by
Zero");
				a = 0; //set a to zero and continue
			}
			System.out.println("a:"+a);
		}
	}
}
```



```
Division by Zero
a:0
a:-6172
a:-6172
a:6172
Division by Zero
a:0
a:-12345
a:324
Division by Zero
a:0
Division by Zero
a:0
a:2057
Division by Zero
a:0
a:-823
a:12345
Division by Zero
a:0
Division by Zero
a:0
Division by Zero
a:0
Division by Zero
a:0
a:6172
a:12345
```

Exception Display

- All exception classes inherit from the Throwable class
- Throwable overrides toString() to describe the exception textually:

```
try
{

}
catch(ArithmeticException e)
{
        System.out.println("Exception:"+e)
}
```

The following text will be displayed:

Exception: java.lang.ArithmeticException: / by zero

Multiple Catch Clauses

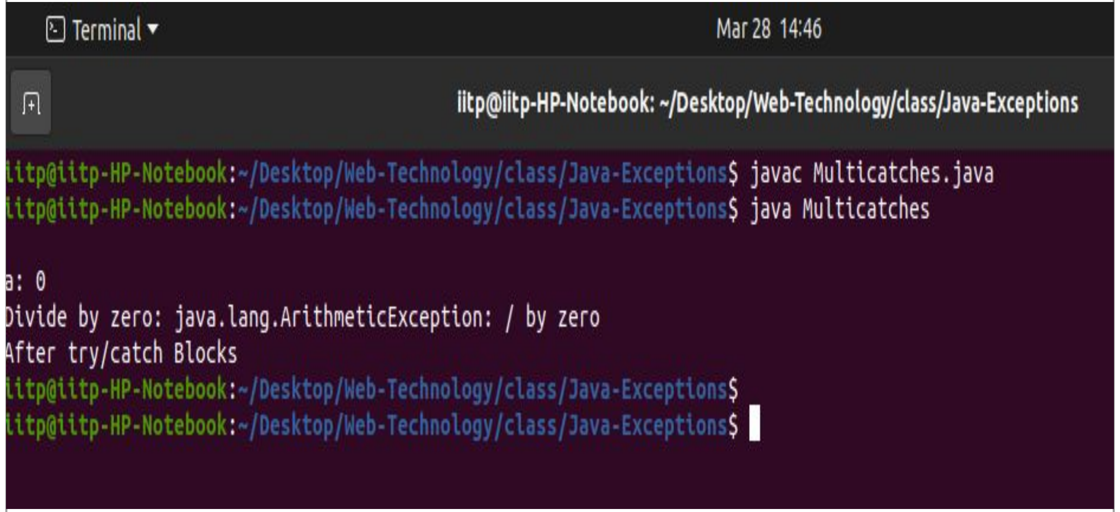
- When more than one exception can be raised by a single piece of code, several catch clauses can be used with one try block:
  - each catch catches a different kind of exception
  - when an exception is thrown, the first one whose type matches that of the exception is executed
  - after one catch executes, the other are bypassed and the execution continues after the try/catch block

Multiple Catch Clauses (Contd.)

```
class Multicatches
{
        public static void main(String args[])
        {
                try
                {
                        int a = args.length;
                        System.out.println("a: "+a);
                        int b = 42/a;
                        int c[] = {1};
                        c[42] = 99;
                }
                catch(ArithmeticException e)
                {
                        System.out.println("Divide by zero: "+e);
                }
                catch(ArrayIndexOutOfBoundsException e)
                {
                        System.out.println("Array Index oob: "+e);
                }
                System.out.println("After try/catch Blocks");
        }
}
```

Multiple Catch Clauses (Contd.)

```
class Multicatches
{
        public static void main(String args[])
        {
                try
                {
                        int a = args.length;
                        System.out.println("a: "+a);
                        int b = 42/a;
                        int c[] = {1};
                        c[42] = 99;
                }
                catch(ArithmeticException e)
                {
                        System.out.println("Divide by zero: "+e);
                }
                catch(ArrayIndexOutOfBoundsException e)
                {
                        System.out.println("Array Index oob: "+e);
                }
                System.out.println("After try/catch Blocks");
        }
}
```


Terminal
Mar 28 14:46
iitp@iitp-HP-Notebook: ~/Desktop/Web-Technology/class/Java-Exceptions

```
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Exceptions$ javac Multicatches.java
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Exceptions$ java Multicatches

a: 0
Divide by zero: java.lang.ArithmeticException: / by zero
After try/catch Blocks
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Exceptions$
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Exceptions$ 
```

Multiple Catch Clauses (Contd.)

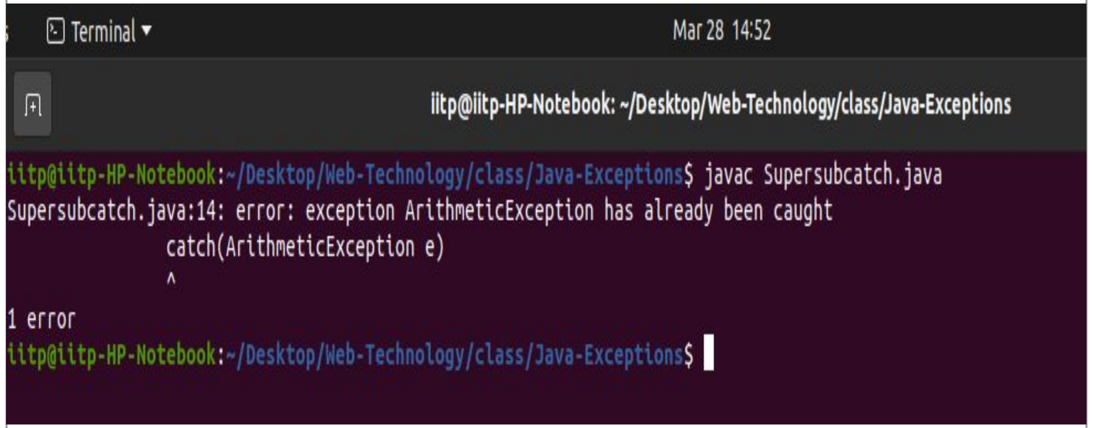
- Order is important:
  - catch clauses are inspected top-down
  - a clause using a super-class will catch all sub-class exceptions
- Therefore, specific exceptions should appear before more general ones. In particular, exception sub-classes must appear before super-classes

Multiple Catch Clauses (Contd.)

```
class Supersubcatch
{
	public static void main(String args[])
	{
		try
		{
			int a = 0;
			int b = 42/a;
		}
		catch(Exception e)
		{
			System.out.println("Generic Exception Catch");
		}
		catch(ArithmeticException e)
		{
			System.out.println("This block is never reached");
		}
	}
}
```

Multiple Catch Clauses (Contd.)

```
class Supersubcatch
{
        public static void main(String args[])
        {
                try
                {
                        int a = 0;
                        int b = 42/a;
                }
                catch(Exception e)
                {
                        System.out.println("Generic Exception
Catch");
                }
                catch(ArithmeticException e)
                {
                        System.out.println("This block is never
reached");
                }
        }
}
```


Terminal
Mar 28 14:52
iitp@iitp-HP-Notebook: ~/Desktop/Web-Technology/class/Java-Exceptions

```
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Exceptions$ javac Supersubcatch.java
Supersubcatch.java:14: error: exception ArithmeticException has already been caught
                catch(ArithmeticException e)
                ^
1 error
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Exceptions$
```

References

1. https://www.javatpoint.com/nested-interface
2. http://etutorials.org/cert/java+certification/Chapter+6.+Object-oriented+Programming/6.7+Polymorphism+and+Dynamic+Method+Lookup/#:~:text=Dynamic%20method%20lookup%20is%20the,instance%20method%20is%20not%20polymorphic.
3. http://coding-guru.com/polymorphism-java/
4. https://coderanch.com/t/378538/java/Dynamic-method-lookup
5. https://www.baeldung.com/java-inner-interfaces
6. https://cs-fundamentals.com/java-programming/java-static-nested-or-inner-interfaces