

OBJECT-ORIENTED LANGUAGE AND THEORY

10. EXCEPTION AND EXCEPTION HANDLER

Nguyen Thi Thu Trang
trangntt@soict.hust.edu.vn



Outline

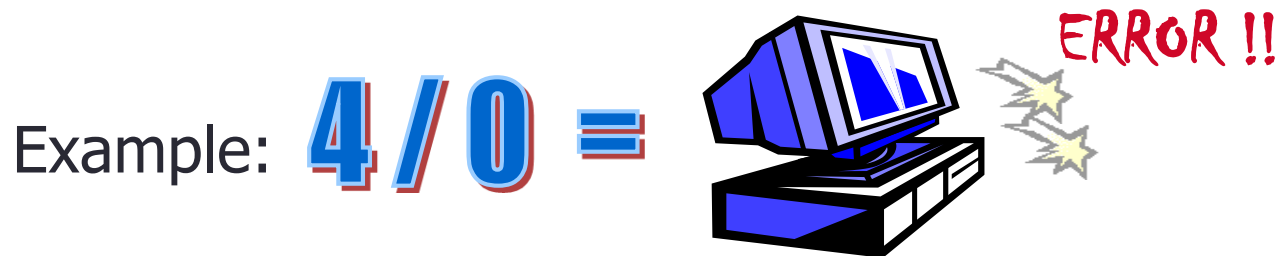


1. Exceptions

2. Catching and handling exceptions
3. Exception delegation
4. User-defined exceptions

1.1. What is exception?

- Exception = Exceptional event
- Definition: An exception is an event that occurs in the **execution** of a program and it **breaks** the expected flow of the program.



1.1. What is exception? (2)

- Exception is an particular error
 - Unexpected results
- When an exception occurs, if it is not handled, the program will exit immediately and the control is returned to the OS



1.2. Classical Error Handler

- Writing handling codes where errors occur
 - Making programs more complex
 - Not always have enough information to handle
 - Some errors are not necessary to handle
- Sending status to upper levels
 - Via arguments, return values or global variables (flag)
 - Easy to mis-understand
 - Still hard to understand

Example

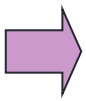
```
int devide(int num, int denom, int *error)
{
    if (denom != 0) {
        *error = 0;
        return num/denom;
    } else {
        *error = 1;
        return 0;
    }
}
```

Disadvantages

- Difficult to control all cases
 - Arithmetic errors, memory errors,...
- Developers often forget to handle errors
 - Human
 - Lack of experience, deliberately ignore

Outline

1. Exceptions



2. Catching and handling exceptions

3. Exception delegation

4. User-defined exceptions

2.1. Goals of exception handling

- Making programs more reliable, avoiding unexpected termination
- Separating blocks of code that might cause exceptions and blocks of code that handle exceptions

.....

IF B IS ZERO GO TO ERROR

C = A/B

PRINT C

GO TO EXIT

ERROR:

DISPLAY "DIVISION BY ZERO"

EXIT:

END

} Error handling block

Separating code

- Classic programming: `readFile()` function: not separate the main logic processing and error handling.

```
errorCodeType 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;  
            }  
        }  
    }  
}
```

Classic Programming

```
    } else {  
        errorCode = -3;  
    }  
    close the file;  
    if (theFileDidntClose && errorCode == 0) {  
        errorCode = -4;  
    } else {  
        errorCode = errorCode and -4;  
    }  
} else {  
    errorCode = -5;  
}  
return errorCode;  
}
```

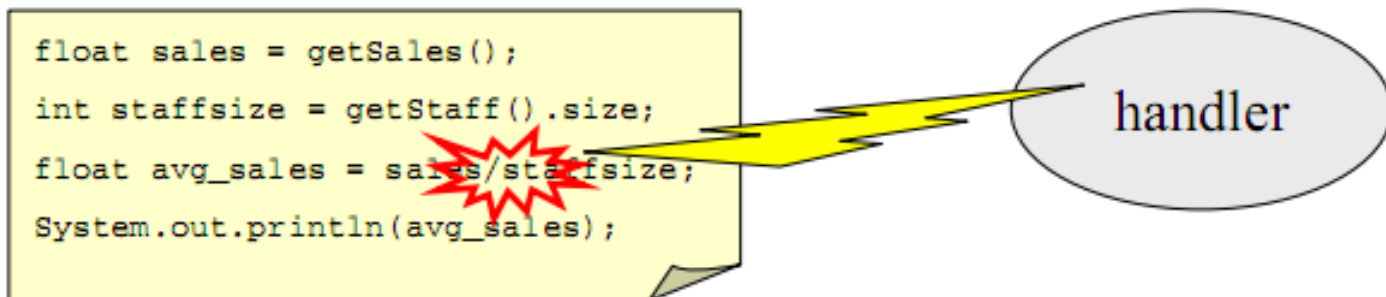
Exception Handling

- Exception mechanism allows focusing on writing code for the main thread and then handling exception in another place

```
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;  
    }  
}
```

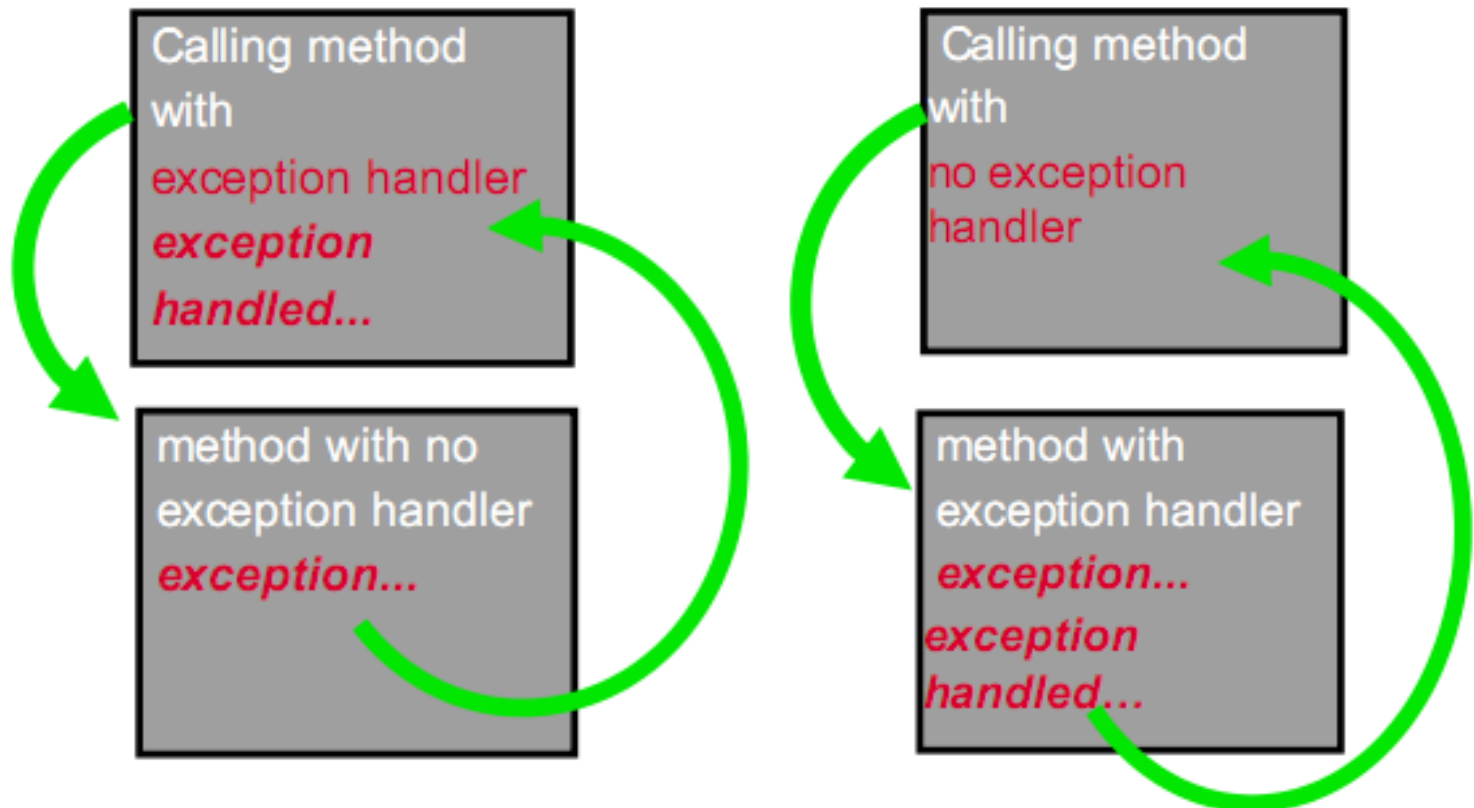
2.2. Models for handling exceptions

- Object oriented approach
 - Packing unexpected conditions in **an object**
 - When an exception occurs, the object corresponding to the exception is created and stores all the detailed information about the exception
 - Providing an efficient mechanism in handling errors
 - Separating irregular control threads with regular threads



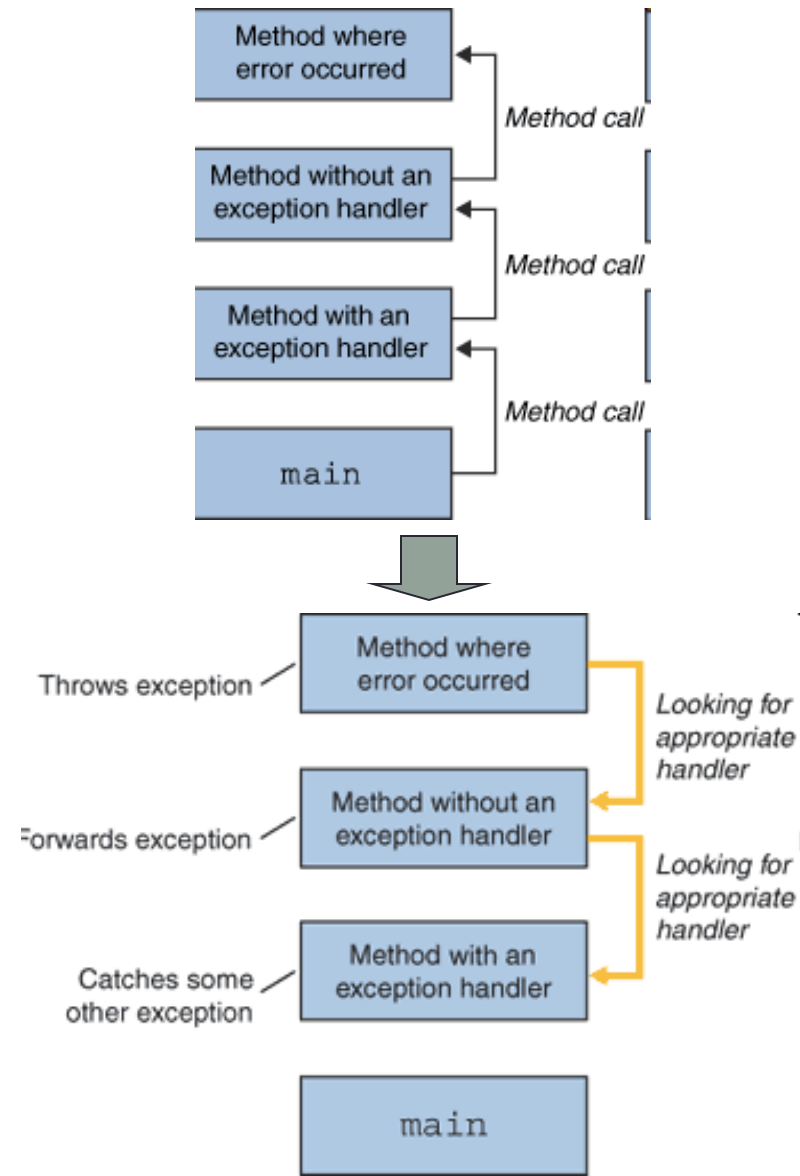
2.2. Models for handling exceptions (2)

- Exceptions need to be handled at the method that causes the exceptions or delegated to its caller method



2.3. Exception handling in Java

- Java has a strong mechanism for handling exceptions
- Exception handling in Java is done via object-oriented model:
 - All the exceptions are representations of a class derived from the class Throwable or its child classes
 - These objects must send the information of exceptions (type and status of the program) from the exceptions place to where they are controlled/handled



2.3. Exception handling in Java (2)

- Key words
 - `try`
 - `catch`
 - `finally`
 - `throw`
 - `throws`

2.3.1. try/catch block

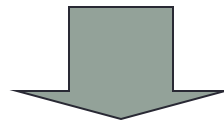
- try ... catch block: Separating the regular block of program and the block for handling exceptions
 - try {...}: Block of code that might cause exceptions
 - catch() {...}: Catching and handling exceptions

```
try {  
    // Code block that might cause exception  
}  
catch (ExceptionType e) {  
    // Handling exception  
}
```

- ❑ **ExceptionType** is a descendant of the **Throwable**

Example of not handling exceptions

```
class NoException {  
    public static void main(String args[]) {  
        String text = args[0];  
        System.out.println(text);  
    }  
}
```



```
D:\FIT-HUT\Lectures\OOP\OOP-Java\Demo>java NoException  
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0  
    at NoException.main<NoException.java:3>  
  
D:\FIT-HUT\Lectures\OOP\OOP-Java\Demo>
```

Example of handling exceptions

```
class ArgExceptionDemo {  
    public static void main(String args[]) {  
        try {  
            String text = args[0];  
            System.out.println(text);  
        }  
        catch (Exception e) {  
            System.out.println("Hay nhap tham so khi chay!");  
        }  
    }  
}
```



```
D:\FIT-HUT\Lectures\OOP\OOP-Java\Demo>java ArgExceptionDemo  
Hay nhap tham so khi chay!  
D:\FIT-HUT\Lectures\OOP\OOP-Java\Demo>_
```

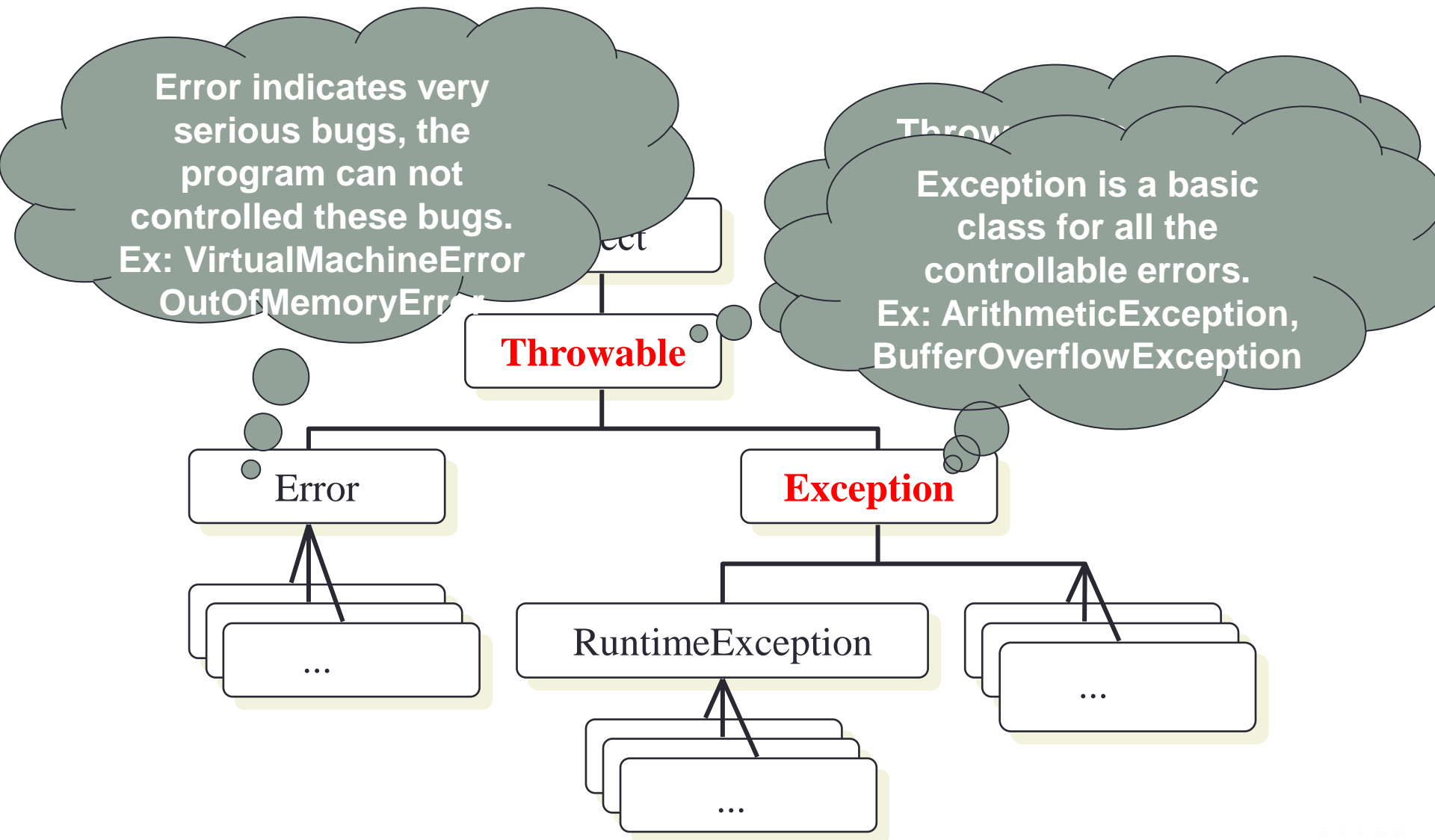
Example of division by 0

```
public class ChiaCho0Demo {  
    public static void main(String args[]) {  
        try {  
            int num = calculate(9,0);  
            System.out.println(num);  
        }  
        catch(Exception e) {  
            System.err.println("Co loi xay ra: " + e.toString());  
        }  
    }  
    static int calculate(int no, int no1){  
        int num = no / no1;  
        return num;  
    }  
}
```



```
Co loi xay ra: java.lang.ArithmeticException: / by zero  
Press any key to continue . . .
```

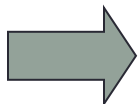
2.3.2. Exception hierarchical tree in Java



a. Class Throwable

- A variable of type `String` to store detailed information about exceptions that already occurred
- Some basic functions
 - `new Throwable(String s)`: Creates an exception and the exception information is `s`
 - `String getMessage()`: Get exception information
 - `String getString()`: Brief description of exceptions
 - `void printStackTrace()`: Print out all the involving information of exceptions (name, type, location...)
 - ...

```
public class StckExceptionDemo {  
    public static void main(String args[]){  
        try {  
            int num = calculate(9,0);  
            System.out.println(num);  
        }  
        catch(Exception e) {  
            System.err.println("Co loi xay ra :"  
                               + e.getMessage());  
            e.printStackTrace();  
        }  
    }  
    static int calculate(int no, int no1)    {  
        int num = no / no1;  
        return num;  
    }  
}
```



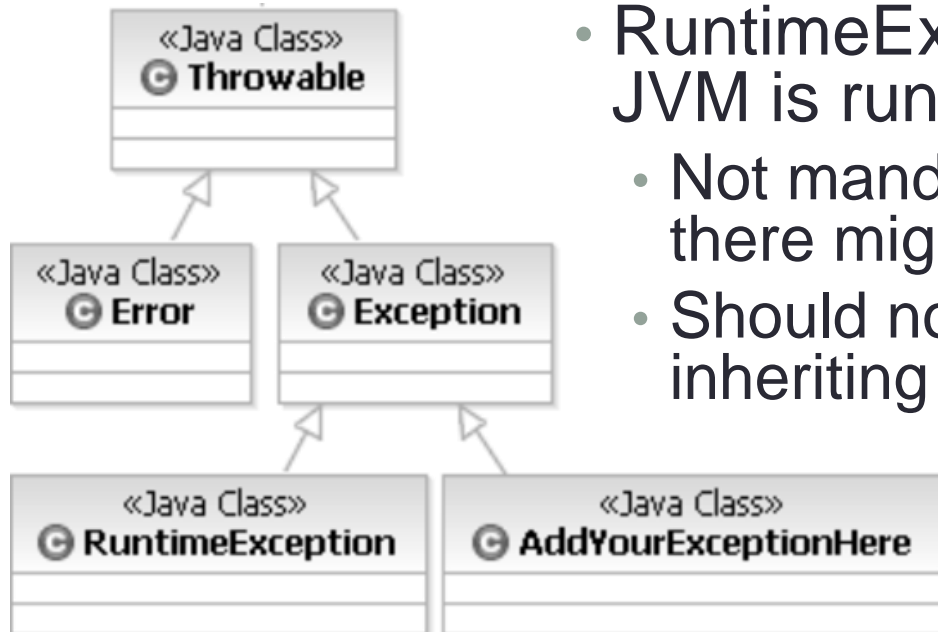
```
Co loi xay ra :/ by zero  
java.lang.ArithmeticException: / by zero  
    at StckExceptionDemo.calculate(StckExceptionDemo.java:14)  
    at StckExceptionDemo.main(StckExceptionDemo.java:4)  
Press any key to continue . . . _
```

b. Class Error

- Contains critical and unchecked exceptions (unchecked exception) because it might occur at many parts of the program.
- Is called un-recoverable exception
- Do not need to check in your Java source code
- Child classes:
 - VirtualMachineError: InternalError, OutOfMemoryError, StackOverflowError, UnknownError
 - ThreadDeath
 - LinkageError:
 - IncompatibleClassChangeError
 - AbstractMethodError, InstantiationException, NoSuchFieldError, NoSuchMethodError...
 - ...
- ...

c. Class Exception

- Has exception types that should/must be caught and handled or delegated.
- Developers might create their own exceptions by inheriting from Exception
- RuntimeException might appear while JVM is running
 - Not mandatory to catch exceptions even there might be errors
 - Should not write your own exception inheriting from this class



Some derived classes of Exception

- ClassNotFoundException, SQLException
- java.io.IOException:
 - FileNotFoundException, EOFException...
- RuntimeException:
 - NullPointerException, BufferOverflowException
 - ClassCastException, ArithmeticException
 - IndexOutOfBoundsException:
 - ArrayIndexOutOfBoundsException,
 - StringIndexOutOfBoundsException...
 - IllegalArgumentException:
 - NumberFormatException, InvalidParameterException...
 - ...

Example of IOException

```
import java.io.InputStreamReader;
import java.io.IOException;
public class HelloWorld{
    public static void main(String[] args) {
        InputStreamReader isr = new
            InputStreamReader(System.in);
        try {
            System.out.print("Nhap vao 1 ky tu: ");
            char c = (char) isr.read();
            System.out.println("Ky tu vua nhap: " + c);
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }
}
```



```
Nhap vao 1 ky tu: b
Ky tu vua nhap: b
Press any key to continue . . .
```

2.3.3. Nested try – catch blocks

- A small part of a code block causes an error, but the whole block cause another error → Need to have nested exception handlers.
- When there are nested try blocks, the inner try block will be done first.

```
try {  
    // May cause IOException  
    try {  
        // May cause NumberFormatException  
    }  
    catch (NumberFormatException e1) {  
        // Handle NumberFormatException  
    }  
} catch (IOException e2) {  
    // Handle IOException  
}
```

2.3.4. Multiple catch block

- A block of code might cause more than one exception
→ Need to use multiple catch block.

```
try {  
    // May cause multiple exception  
} catch (ExceptionType1 e1) {  
    // Handle exception 1  
} catch (ExceptionType2 e2) {  
    // Handle exception 2  
} ...
```

- **ExceptionType1** must be a derived class or an level-equivalent class of the class **ExceptionType2** (in the inheritance hierarchy tree)

- ExceptionType1 must be a derived class or an level-equivalent class of the class ExceptionType2 (in the inheritance hierarchy tree)

```
class MultipleCatch1 {
    public static void main(String args[])
    {
        try {
            String num = args[0];
            int numValue = Integer.parseInt(num);
            System.out.println("Dien tich hv la: "
                               + numValue * numValue);
        } catch (Exception e1) {
            System.out.println("Hay nhap canh cua
hv!");
        } catch (NumberFormatException e2) {
            System.out.println("Not a number!");
        }
    }
}
```


Error

D:\exception java.lang.NumberFormatException
has already been caught

- `ExceptionType1` must be a derived class or an level-equivalent class of the class `ExceptionType2` (in the inheritance hierarchy tree)

```
class MultipleCatch1 {  
    public static void main(String args[])  
    {  
        try {  
            String num = args[0];  
            int numValue = Integer.parseInt(num);  
            System.out.println("Dien tich hv la: "  
                               + numValue * numValue);  
        } catch (ArrayIndexOutOfBoundsException e1) {  
            System.out.println("Hay nhap canh cua hv!");  
        } catch (NumberFormatException e2) {  
            System.out.println("Hay nhap 1 so!");  
        }  
    }  
}
```

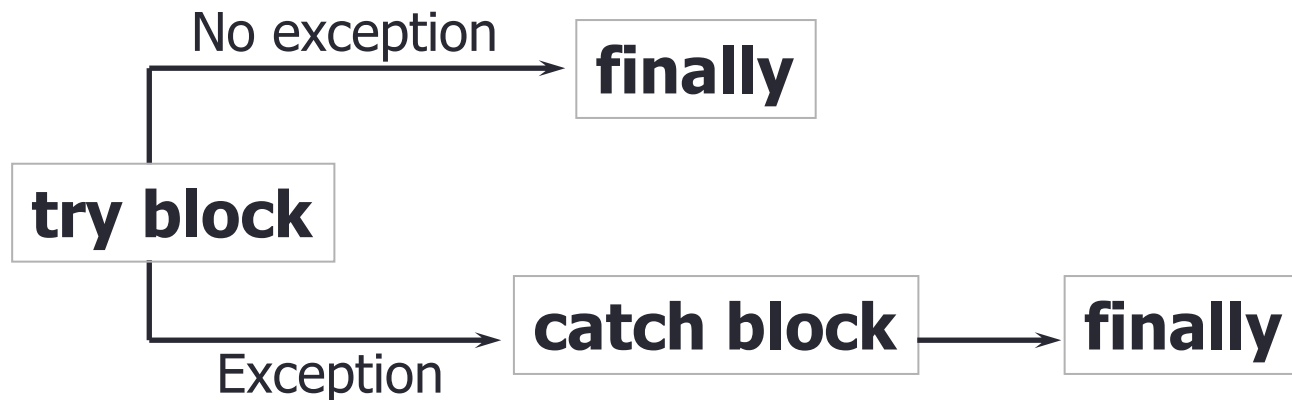
```
public static void main( String args[]) {  
    try {  
        // format a number  
        // read a file  
        // something else...  
    }  
    catch(IOException e) {  
        System.out.println("I/O error "+e.getMessage());  
    }  
    catch(NumberFormatException e) {  
        System.out.println("Bad data "+e.getMessage());  
    }  
    catch(Throwable e) { // catch all  
        System.out.println("error: " + e.getMessage());  
    }  
}
```



```
...  
public void openFile() {  
    try {  
        // constructor may throw FileNotFoundException  
        FileReader reader = new FileReader("someFile");  
        int i=0;  
        while(i != -1) {  
            //reader.read() may throw IOException  
            i = reader.read();  
            System.out.println((char) i );  
        }  
        reader.close();  
        System.out.println("--- File End ---");  
    } catch (FileNotFoundException e) {  
        //do something clever with the exception  
    } catch (IOException e) {  
        //do something clever with the exception  
    }  
}  
...
```

2.3.5. finally block

- Ensure that every necessary tasks are done when an exception occurs
 - Closing file, closing socket, connection
 - Releasing resource (if neccessary)...
- Must be done even there is an exception occurring or not.



The syntax try ... catch ... finally

```
try {  
    // May cause exceptions  
}  
catch (ExceptionType e) {  
    // Handle exceptions  
}  
finally {  
    /* Necessary tasks for all cases:  
    exception is raised or not */  
}
```

- ❑ If there is a block try, there must be a block catch or a block finally or both

```
class StrExceptionDemo {
    static String str;
    public static void main(String s[]) {
        try {
            System.out.println("Before exception");
            staticLengthmethod();
            System.out.println("After exception");
        }
        catch (NullPointerException ne) {
            System.out.println("There is an error");
        }
        finally {
            System.out.println("In finally");
        }
    }

    static void staticLengthmethod() {
        System.out.println(str.length());
    }
}
```

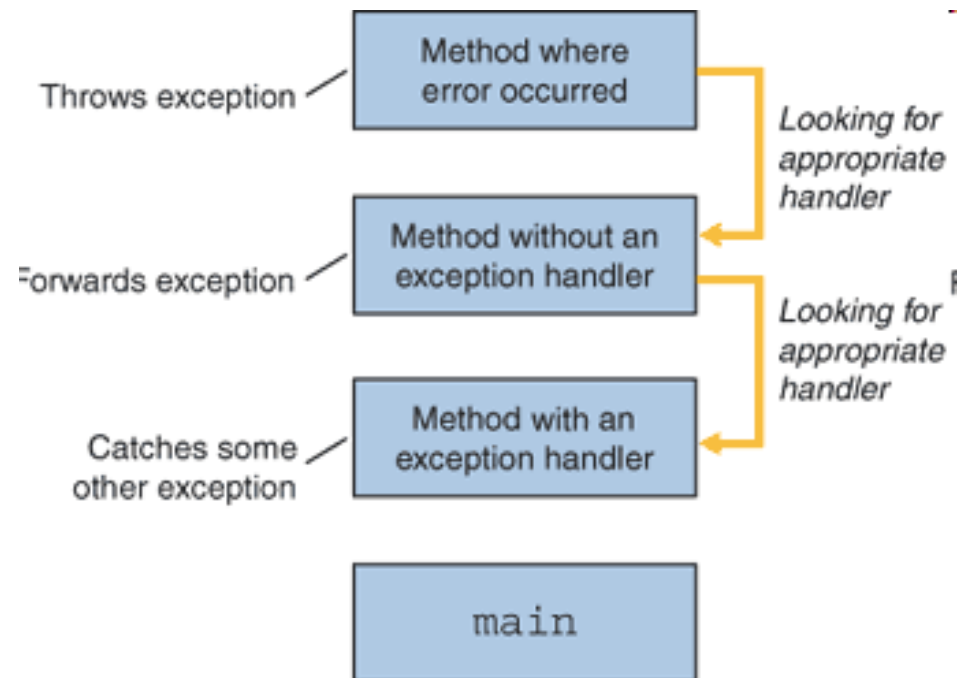
```
public void openFile() {
    try {
        // constructor may throw FileNotFoundException
        FileReader reader = new FileReader("someFile");
        int i=0;
        while(i != -1) {
            //reader.read() may throw IOException
            i = reader.read();
            System.out.println((char) i );
        }
    } catch (FileNotFoundException e) {
        //do something clever with the exception
    } catch (IOException e) {
        //do something clever with the exception
    } finally {
        reader.close();
        System.out.println("--- File End ---");
    }
}
```

Outline

1. Exceptions
2. Catching and handling exceptions
- ➔ 3. Exception delegation
4. User-defined exceptions

Two ways to deal with exceptions

- Handle immediately
 - Using the block try ... catch (finally if necessary).
- Delegating to its caller:
 - If we don't want to handle immediately
 - Using throw and throws



3.1. Exception delegation

- A method can delegate exceptions to its caller:
 - Using **throws** at the method definition to tell its caller of ExceptionType that it might cause an exception ExceptionType
 - Using **throw** anExceptionObject in the body of function in order to throw an exception when necessary

- For example

```
public void myMethod(int param) throws Exception{  
    if (param < 10) {  
        throw new Exception("Too low!");  
    }  
    //Blah, Blah, Blah...  
}
```


3.1. Exception delegation (2)

- If a method has some code that throws an exception, its declaration must declare a “throw” of that exception or the parent class of that exception

```
public void myMethod(int param) {  
    if (param < 10) {  
        throw new Exception("Too low!");  
    }  
    //Blah, Blah, Blah...  
}
```

→ unreported exception `java.lang.Exception`; must be caught or declared to be thrown

3.1. Exception delegation (3)

- A method without exception declaration will throw RuntimeException because this exception is delegated to JVM
- Example

```
class Test {  
    public void myMethod(int param) {  
        if (param < 10) {  
            throw new RuntimeException("Too low!");  
        }  
        //Blah, Blah, Blah...  
    }  
}
```

3.1. Exception delegation (3)

- At the caller of the method that has exception delegation (except `RuntimeException`):
 - Or the caller method must delegate to its caller
 - Or the caller method must catch the delegated exception (or its parent class) and handle immediately by `try... catch` (`finally` if necessary)

```
public class DelegateExceptionDemo {  
    public static void main(String args[]) {  
        int num = calculate(9,3);  
        System.out.println("Lan 1: " + num);  
        num = calculate(9,0);  
        System.out.println("Lan 2: " + num);  
    }  
    static int calculate(int no, int no1)  
        throws ArithmeticException {  
        if (no1 == 0)  
            throw new  
                ArithmeticException("Cannot devide by 0!");  
        int num = no / no1;  
        return num;  
    }  
}
```

```

public class DelegateExceptionDemo {
    public static void main(String args[]){
        int num = calculate(9,3);
        System.out.println("Lan 1: " + num);
        num = calculate(9,0);
        System.out.println("Lan 2: " + num);
    }
    static int calculate(int no, int no1)
        throws Exception {
        if (no1 == 0)
            throw new
                ArithmeticException("Cannot divide by 0!");
        int num = no / no1;
        return num;
    }
}

```



G:\Java Example\DelegateExceptionDemo.java:3: unreported exception java.lang.Exception;
must be caught or declared to be thrown

```

        int num = calculate(9,3);
                        ^

```

G:\Java Example\DelegateExceptionDemo.java:5: unreported exception java.lang.Exception;
must be caught or declared to be thrown

```

        num = calculate(9,0);

```

```
public class DelegateExceptionDemo {
    public static void main(String args[]){
        try {
            int num = calculate(9,3);
            System.out.println("Lan 1: " + num);
            num = calculate(9,0);
            System.out.println("Lan 2: " + num);
        } catch(Exception e) {
            System.out.println(e.getMessage());
        }
    }
    static int calculate(int no, int no1)
        throws Exception {
        if (no1 == 0)
            throw new
                ArithmeticException("Cannot devide by 0!");
        int num = no / no1;
        return num;
    }
}
```

3.1. Exception delegation (4)

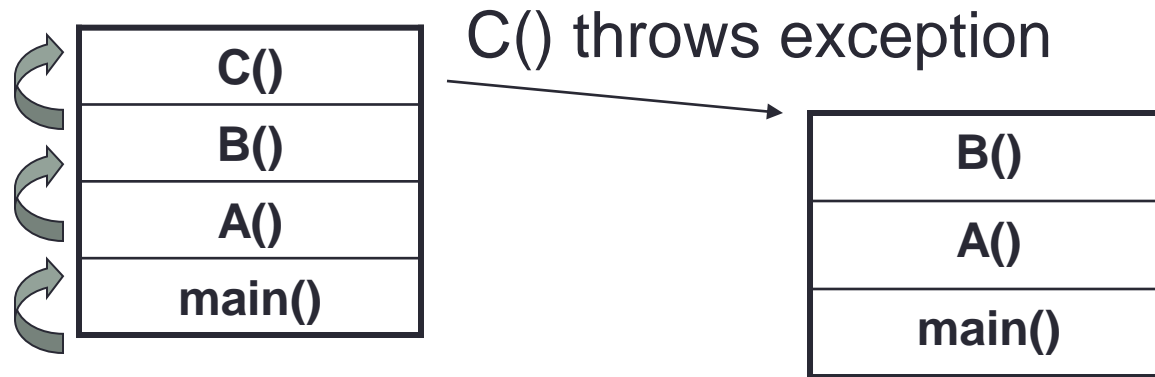
- A method can delegate more than 1 exception

```
public void myMethod(int age, String name)
    throws ArithmeticException, NullPointerException{
    if (age < 18) {
        throw new ArithmeticException
            ("Age must be at least 18");
    }
    if (name == null) {
        throw new NullPointerException
            ("Name must be provided");
    }
    //Blah, Blah, Blah...
}
```

3.2. Exception propagation

- Scenario:
 - Assuming that in `main()` method `A()` is called, `B()` is called in `A()`, `C()` is called in `B()`. Then a stack of method is created.
 - Assuming that in `C()` there is an exception occurring.

3.2. Exception Propagation (2)



If `C()` has an error and throws an exception but in `C()` that exception is not handled, hence there is only one place that handles the exception, that place is where `C()` is called, it is the method `B()`.

If in `B()` there is no exception handling, then the exception must be handled in `A()` ... This is called Exception Propagation

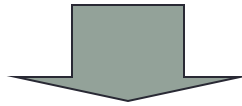
If in `main()`, the exception thrown from `C()` can not be handled, the program will be interrupted.

3.3. Inheritance and exception delegation

- When overriding a method of a parent class, methods in its child classes can not throw any new exception
- Overriden method in a child class can only throw a set of exceptions that are/similar to/ a subset of exceptions thrown from the parent class.

3.3. Inheritance and exception delegation(2)

```
class Disk {  
    void readFile() throws EOFException {}  
}  
class FloppyDisk extends Disk {  
    void readFile() throws IOException {} // ERROR!  
}
```



```
class Disk {  
    void readFile() throws IOException {}  
}  
class FloppyDisk extends Disk {  
    void readFile() throws EOFException {} //OK  
}
```

3.4. Advantages of exception delegation

- Easy to use
 - Making programs easier to read and more reliable
 - Easy to send control to the places that can handle exceptions
 - Can throw many types of exceptions
- Separating exception handling from the main code
- Do not miss any exception (throw automatically)
- Grouping and categorizing exceptions
- Making program easier to read and more reliable

Outline

1. Exceptions
2. Catching and handling exceptions
3. Exception delegation
- 4. User-defined exceptions

4. User-defined exception

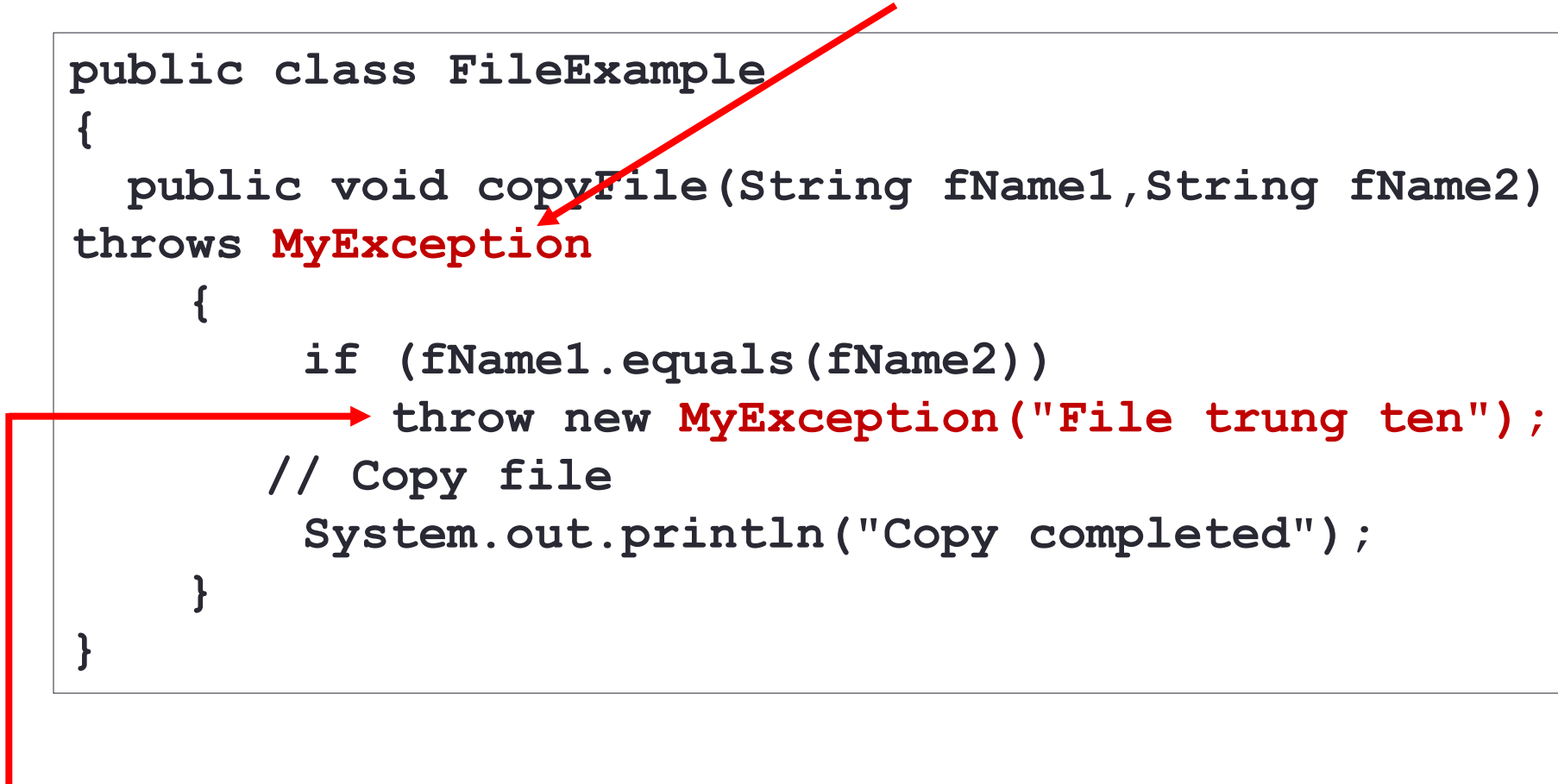
- Exceptions provided can not controll all the errors → Need to have exceptions that are defined by users.
 - Inheriting from the class **Exception** or one of its child classes
 - Having all the methods of the class **Throwable**

```
public class MyException extends Exception {  
    public MyException(String msg) {  
        super(msg) ;  
    }  
    public MyException(String msg, Throwable cause) {  
        super(msg, cause) ;  
    }  
}
```

Using self-defined exceptions

Declaring that an exception might be thrown

```
public class FileExample
{
    public void copyFile(String fName1,String fName2)
    throws MyException
    {
        if (fName1.equals(fName2))
            throw new MyException("File trung ten");
        // Copy file
        System.out.println("Copy completed");
    }
}
```

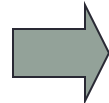
A red arrow originates from the **MyException** declaration in the `throws` clause and points to the `throw new MyException` statement within the `copyFile` method. A second red arrow originates from the `throw` statement and points down to the **Throwing an exception** text at the bottom of the slide.

Throwing an exception

Using self-defined exceptions

- Catching and handling exceptions

```
public class Test {  
    public static void main(String[] args) {  
        FileExample obj = new FileExample();  
        try {  
            String a = args[0];  
            String b = args[1];  
            obj.copyFile(a,b);  
        } catch (MyException e1) {  
            System.out.println(e1.getMessage());  
        }  
        catch(Exception e2) {  
            System.out.println(e2.toString());  
        }  
    }  
}
```



```
C:\>java Test a1.txt a1.txt  
File trung ten  
  
C:\>java Test  
java.lang.ArrayIndexOutOfBoundsException: 0
```


Quiz

Modify the following source code so that `copyFile()` method will throw 2 exceptions:

- `MyException` if the 2 file names are equal, and
- `IOException` if there is any error during the copy file process

```
public class FileExample {  
    public void copyFile(String fName1,String fName2)  
        throws MyException{  
        if (fName1.equals(fName2))  
            throw new MyException("Duplicate file name");  
  
        // Copy file  
  
        System.out.println("Copy completed");  
    }  
}
```

Conclusion (3)

- Types of exception handling:
 - Fix errors and call again the method that caused these errors
 - Fix errors and continue running the method
 - Handling differently instead of ignoring the result
 - Exit the program