OBJECT-ORIENTED LANGUAGE AND THEORY

10. EXCEPTION AND EXCEPTION HANDLER

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

1

Outline

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

2

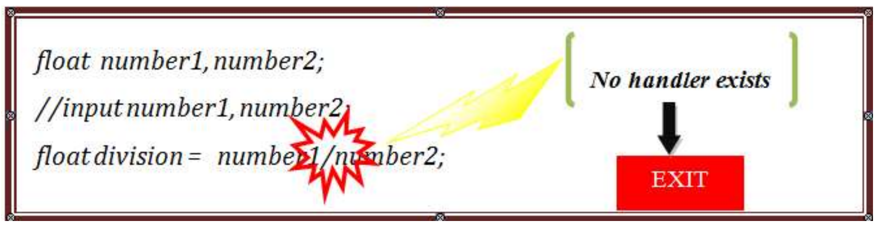
1.1. What is exception?

- 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.

Example: **4 / 0 =**

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



```
float number1, number2;
//input number1, number2
float division =  number1/number2;
```

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

5

Example

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

6

Disadvantages

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

7

Outline

8

2.1. Goals of exception handling

- Making programs more reliable, avoiding un-expected 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

9

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;
            }
```

10

Classic Programming

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

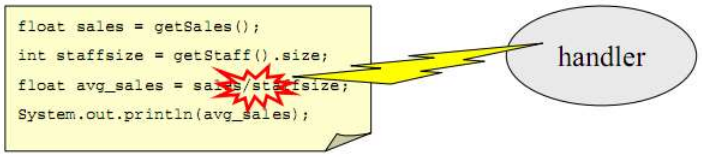
Exception Handling

- 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

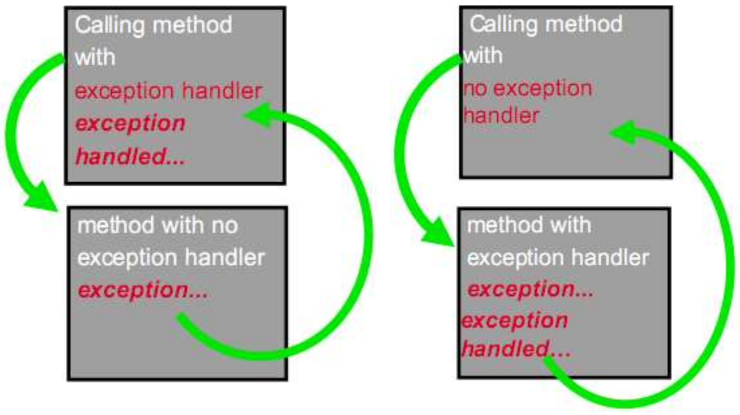


```
float sales = getSales();
int staffsize = getStaff().size;
float avg_sales = sales/staffsize;
System.out.println(avg_sales);
```

13

2.2. Models for handling exceptions (2)

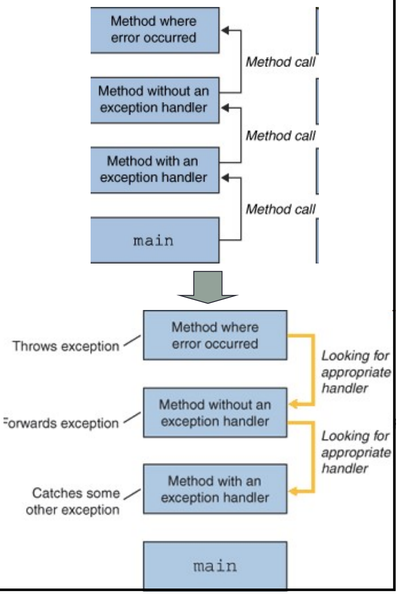
- Exceptions need to be handled at the method that causes the exceptions or delegated to its caller method


Calling method with exception handler exception handled...
method with no exception handler exception...
Calling method with no exception handler
method with exception handler exception... exception handled...

14

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


Method where error occurred
Method call
Method without an exception handler
Method call
Method with an exception handler
Method call
main
Throws exception
Method where error occurred
Looking for appropriate handler
Forwards exception
Method without an exception handler
Looking for appropriate handler
Catches some other exception
Method with an exception handler
main

15

2.3. Exception handling in Java (2)

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

16

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

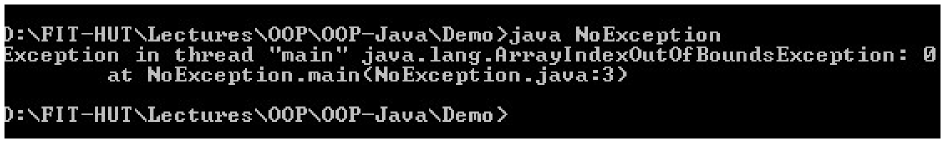
```
try {
     // Code block that might cause exception
}
catch (ExceptionType e) {
     // Handling exception
}
```

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

17

Example of not handling exceptions

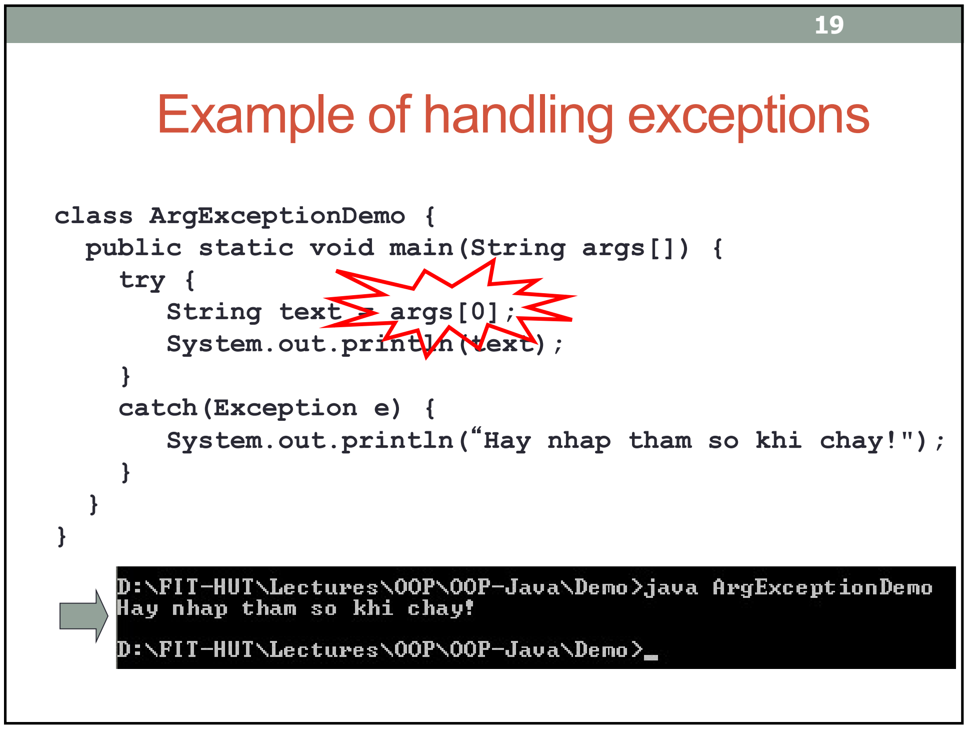
```
class NoException {
 public static void main(String args[]) {
         String text = args[0];
         System.out.println(text);
 }
}
```



```
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>
```

18



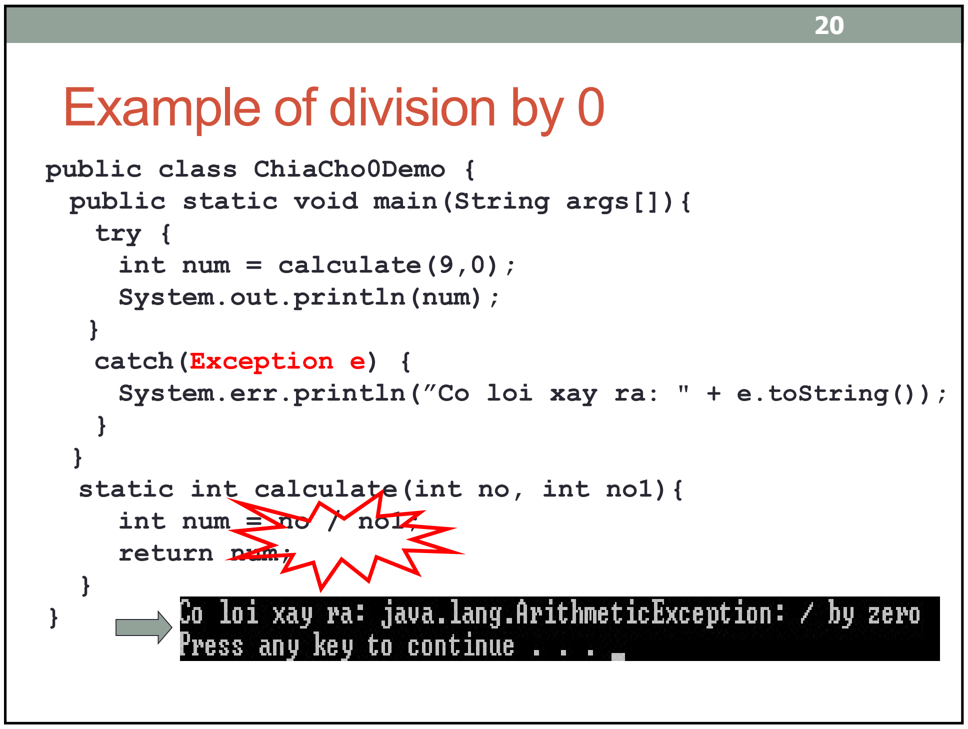
Example of handling exceptions

```
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>_
```

19



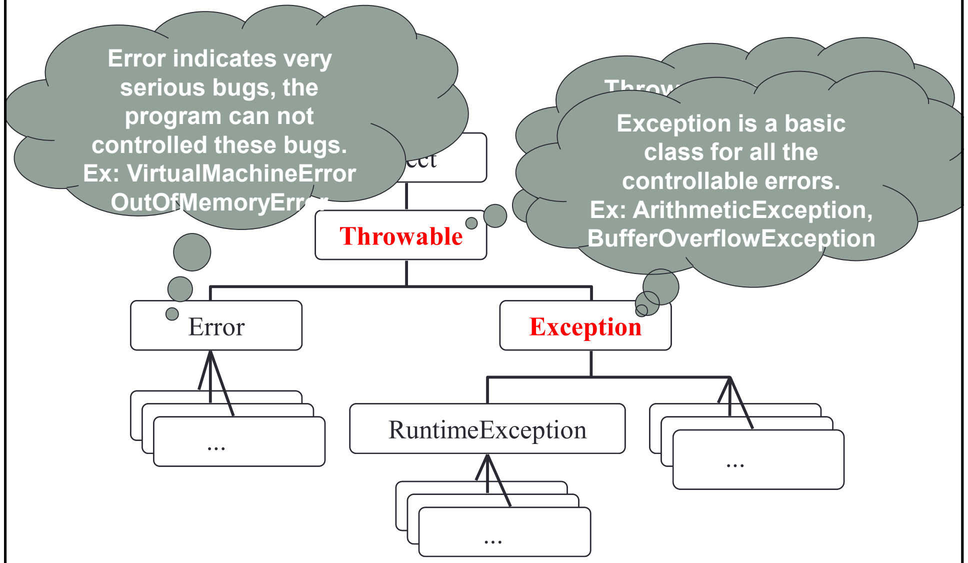
Example of division by 0

```
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 . . . _
```

20

2.3.2. Exception hierarchical tree in Java


Error indicates very serious bugs, the program can not controlled these bugs. Ex: VirtualMachineError OutOfMemoryError
Exception is a basic class for all the controllable errors. Ex: ArithmeticException, BufferOverflowException
Throwable
Error
Exception
RuntimeException
...
...
...

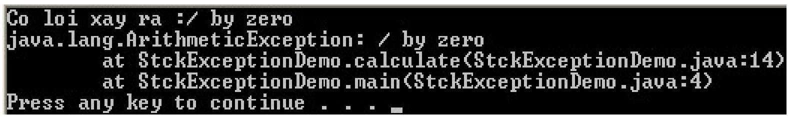
21

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...)
  - …

22

```
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 . . . _
```

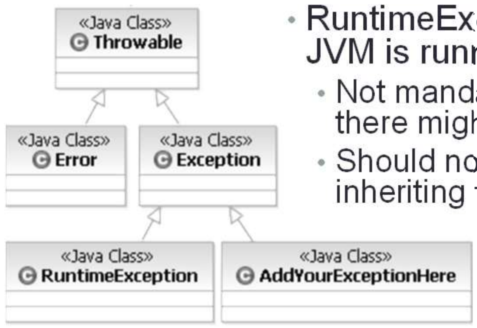
23

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, InstantiationError, NoSuchFieldError, NoSuchMethodError…
    - …
  - …

24

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

25

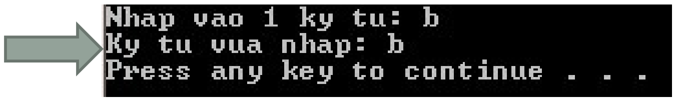
Some derived classes of Exception

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

26

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 . . .
```

27

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
}
```

28

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)

29

- 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

30

- 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!");
   }
  }
}
```

31

```
class MultiCatch2 {
  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();}
   }
  }
}
```

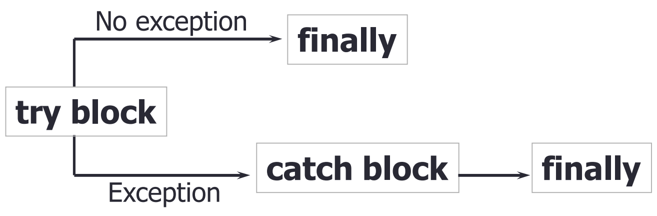
32

```
...
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
  }
}
...
```

33

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.



34

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

35

```
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());
  }
}
```

36

```
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 ---");
  }
}
```

37

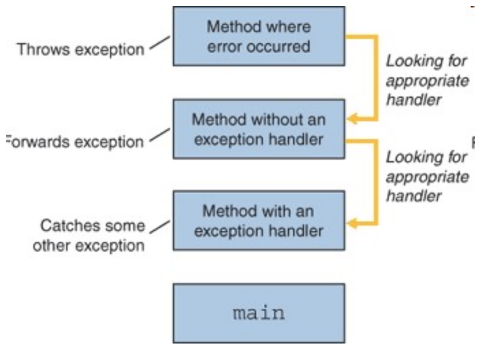
Outline

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

38

Two ways to deal with exceptions

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


Throws exception
Method where error occurred
Looking for appropriate handler
Forwards exception
Method without an exception handler
Looking for appropriate handler
Catches some other exception
Method with an exception handler
main

39

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...
}
```

40

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

41

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...
 }
}
```

42

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)

43

```
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;
  }
}
```

44

```
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;
  }
}
```

Compile

```
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);
```

45

```
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;
  }
}
```

46

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...
}
```

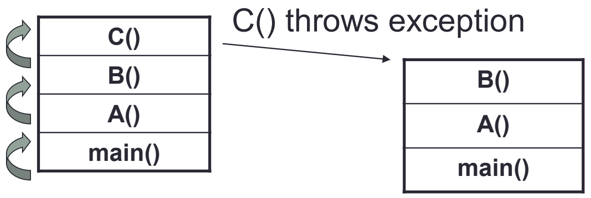
47

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.

48

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.

49

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.

50

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
}
```

51

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

52

52

Outline

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

53

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);
 }
}
```

54

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");
     }
}
```

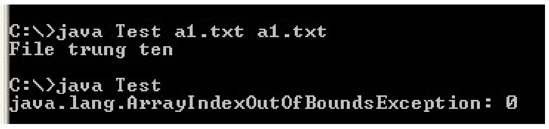
Throwing an exception

55

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
```

56

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");
  }
}
```

57

Conclusion

- Anytime there is an error while running the program, an exception appears.
- All the exceptions must be handled to avoid unexpected termination of the program.
- Handling exceptions allows to handle all the exception in a place.
- Java uses the block try/catch to manage exceptions.

58

Conclusion (2)

- Code blocks in the block try throw exception, and the exception handling must be done in the block catch.
- Many blocks of catch can be used to handle separately different exceptions.
- The keyword throws is used to list all the exceptions that a method can throw.
- The keyword throw is used to throw an exception.
- The block finally performs necessary tasks even there is an exception occurring or not.

59

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

60

Outline

5.1. Assertion là gì?

- Assertion cho phép lập trình viên kiểm tra các giả thiết về chương trình.
  - Trong chương trình giả lập hệ thống giao thông, bạn muốn khẳng định rằng tốc độ dương nhưng nhỏ hơn một giá trị giới hạn nào đó.
- Một assertion chứa một biểu thức boolean mà bạn tin rằng sẽ đúng khi thực hiện – nếu không đúng hệ thống sẽ ném ra một lỗi
  - Bằng việc kiểm tra biểu thức boolean là đúng, assertion xác nhận giả thuyết của bạn, và giúp bạn tự tin hơn rằng chương trình không có lỗi.

5.2. Sử dụng Assertion

- assert expression;
  - expression trả về kiểu boolean, nếu giá trị của nó là false thì hệ thống sẽ tung ra AssertionError.
  - → Không thể thu được bất cứ thông tin gì về lỗi đã xảy ra.
- assert expression1:expression2;
  - expression1 trả về giá trị boolean, biểu thức expression2 có bất kỳ kiểu giá trị nào ngoại trừ lời gọi phương thức trả về kiểu void.
  - Nếu expression1 trả về false
    - Hệ thống tung ra AssertionError.
    - Giá trị trong expression2 sẽ được truyền vào hàm tạo của lớp AssertionError và giá trị đó sẽ được hiển thị để thông báo lỗi.

63

5.2. Sử dụng Assertion

- Việc kiểm tra assertion mặc định bị disable
  - Cần được enable lên sử dụng câu lệnh enableassertions
  - Nếu không được enable thì câu lệnh assertion sẽ không được thực hiện.

64

5.2. Lợi ích của Assertion

- Nhanh và hiệu quả để tìm ra lỗi và sửa lỗi
- Ghi lại các công việc bên trong của chương trình của bạn, giúp nâng cao tính bảo trì
- Lập trình theo thiết kế
  - Các tiền điều kiện (Pre-conditions)
    - Đảm bảo các tiền điều kiện như yêu cầu của khách hàng
  - Các hậu điều kiện (Post-conditions)
    - Đảm bảo hậu điều kiện là kết quả của phương thức gọi
  - Các bất biến bên trong
    - Lập trình viên sử dụng để xác nhận giả thuyết của mình

65

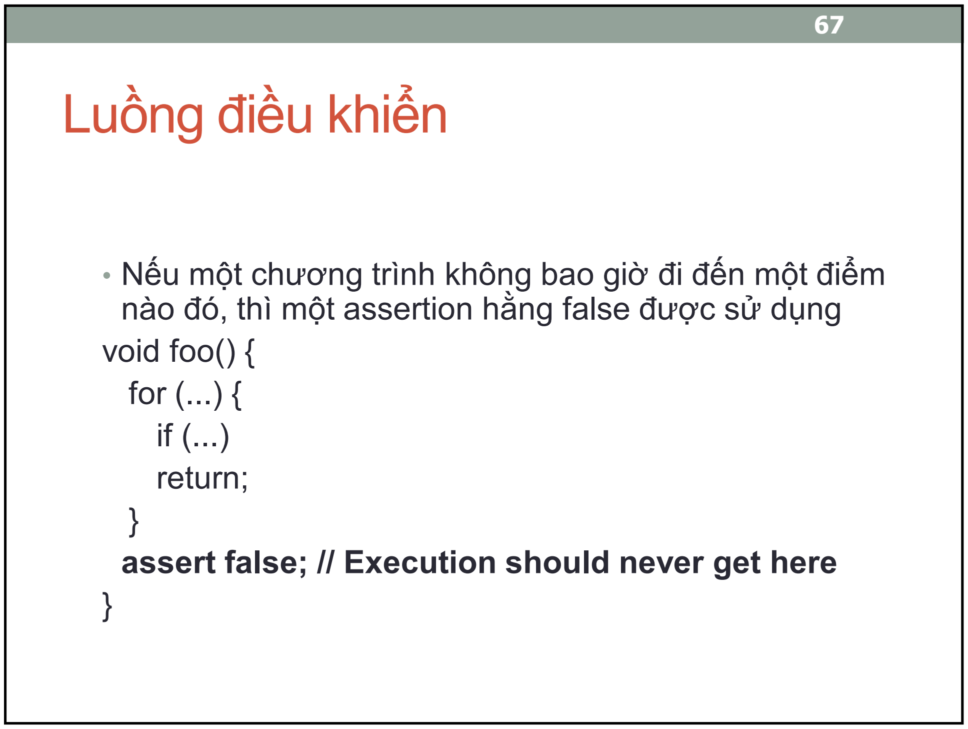
5.2. Lợi ích của Assertion (2)

- Ví dụ:

```
if (i % 3 == 0) { ... }
else if (i % 3 == 1) { ... }
else { // We know (i % 3 == 2)
... }

if (i % 3 == 0) { ... }
else if (i % 3 == 1) { ... }
else { assert i % 3 == 2 : i; ... }
```

66



Luồng điều khiển

- Nếu một chương trình không bao giờ đi đến một điểm nào đó, thì một assertion hằng false được sử dụng

```
void foo() {
  for (...) {
    if (...)
    return;
  }
  assert false; // Execution should never get here
}
```