1495 UNIVERSITY OF ABERDEEN

JC2002 Java Programming

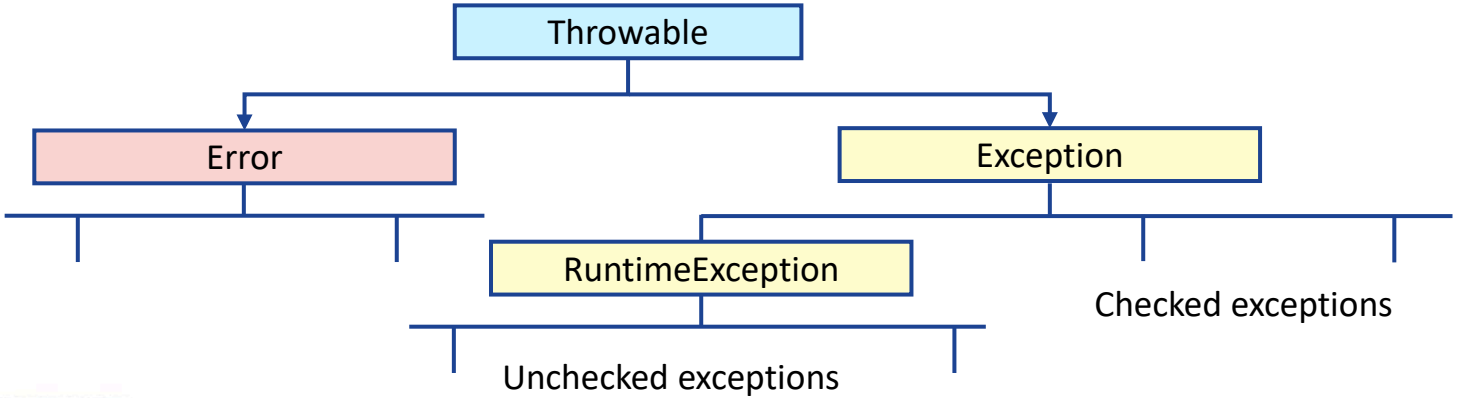
Lecture 17: Exception handling in Java

Exception handling

- An ***exception*** is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
- An ***exception handler*** is a block of code that can handle the exception:
  - Java allows to separate exception handling code from the normal code to improve the readability.
  - Exceptions are propagated across the call stack until exception handler is found so developers can choose at which level exceptions should be handled.
  - Each organisation will have its own house style on how to write and handle exceptions.

Exception handling

- **`Throwable`** class on top, with **`Error`** and **`Exception`** subclasses
- `Errors` and `Exceptions` are further divided in subclasses
  - `Errors` indicate more serious problems that usually cannot be solved at runtime (for example, ***out of memory***).



Error example (infinite recursion)

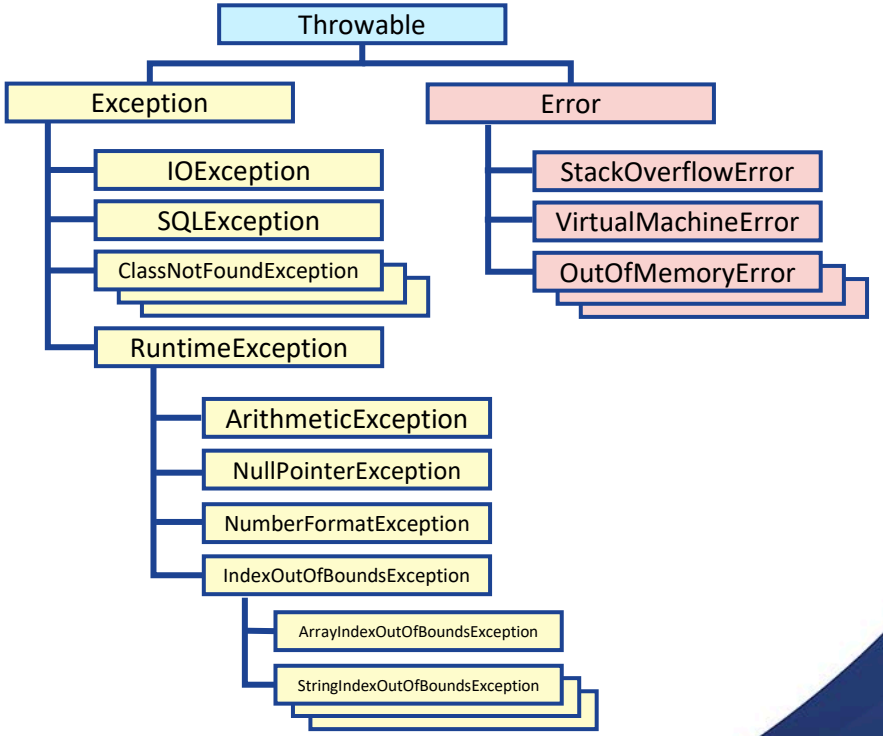
```java
public class SimpleRecursion2 {
    public static void recursiveLoop(int i, int max){
        System.out.print(i + " ");
        if(i < max) {
            recursiveLoop(i, max);
        }
    }
    public static void main(String[] args){
        recursiveLoop(1,10);
        System.out.println();
    }
}
```

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
Exception in thread "main" java.lang.StackOverflowError
```

Index variable **i** not changing, so eventually the stack overflows!

Exception categories

- There are predefined exceptions to cover nearly all possible error situations in practical Java programs.
- It is also possible to create custom exceptions by subclassing the existing classes.
  - The hierarchy of exceptions is not fixed.



Checked and unchecked exceptions

- ***Checked exceptions*** happen at compile time when the source code is transformed into an executable code. ***Unchecked exceptions*** happen at runtime when the executable program starts running.
- ***Checked exceptions*** are all subclasses of **`Exception`**, except the **`RuntimeException`** subclasses, which are ***unchecked exceptions***.
  - Unchecked exceptions are typically result of a programming problem.
  - Many programmers argue against catching unchecked exceptions as these cannot be predicted and if they happen, they point towards a bad code design that should be fixed to prevent the error.
- Checked exceptions *must be* declared by the **`throws`** keyword; otherwise, the compiler will return an error.

Checked exception example

```
1   import java.io.*;
2   public class ExceptionTest1 {
3       public static void main(String args[]) {
4           FileInputStream inputStream = null;
5           inputStream = new FileInputStream("file.txt");
6           int m;
7           while ((m = inputStream.read()) != -1) {
8               System.out.print((char) m);
9           }
10          inputStream.close();
11      }
12  }
```

FileNotFoundException *could* be thrown here!

```
$ javac ExceptionTest1.java
ExceptionTest1.java:5: error: unreported exception FileNotFoundException;
must be caught or declared to be thrown
$
```

Checked exception example with throws

```java
import java.io.*;
public class ExceptionTest2 {
    public static void main(String args[]) throws IOException {
        FileInputStream inputStream = null;
        inputStream = new FileInputStream("file.txt");
        int m;
        while ((m = inputStream.read()) != -1) {
            System.out.print((char) m);
        }
        inputStream.close();
    }
}
```

This allows compiling the code

```
$ javac ExceptionTest2.java
$ java ExceptionTest2
Exception in thread "main" java.io.FileNotFoundException: file.txt (No such
file or directory)
$
```

Unchecked exception example

```java
import java.util.*;
public class ExceptionTest3 {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Give x: "); int x = input.nextInt();
        System.out.print("Give y: "); int y = input.nextInt();
        System.out.println("x / y = " + x/y);
    }
}
```

```
$ javac ExceptionTest3.java
$ java ExceptionTest3
Give x: 10
Give y: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
$
```

This throws an exception (division by zero), if y=0!

Exception handling with try … catch

- By default, the program stops when exception is thrown
- However, it is possible to handle exceptions using **`try…catch`** structure:

This is only run if there was an exception

Program continues here, whether there was an exception or not

```
try {
    do something that can cause an exception
}
catch(Exception e) {
    do this if there was an exception
}
continue the program here normally
```

Example of try ... catch

```
1  import java.util.*;
2  public class TryCatchTest {
3      public static void main(String[] args) {
4          Scanner input = new Scanner(System.in);
5          System.out.print("Give x: "); int x = input.nextInt();
6          System.out.print("Give y: "); int y = input.nextInt();
7          try {
8              System.out.println("x / y = " + x/y);
9          }
10         catch(Exception e) {
11             System.out.println("y can't be zero!");
12         }
13     }
14 }
```

```
Give x: 10
Give y: 0
y can’t be zero!
```

Exception handler

This throws an exception (division by zero), if y=0!

Using keyword throw

- In the previous examples, exceptions were thrown by JVM
- You can also `throw` a new `Exception` object within a method

```
public class Person {
  protected int age
  public void setAge(int age) {
    if(age < 0) {
      throw new IllegalArgumentException("Age can't be negative!");
    }
    this.age = age;
  }
}
```

Using keyword finally

- The `finally` block is often used as a place to release resources acquired in the try block (e.g., database connections, opened files)
- The `finally` block is guaranteed to execute unless the try block or catch block call `System.exit()` which stops the Java interpreter
- Avoid placing code that can `throw` an exception in a `finally` block
- If such code is required, enclose the code in a `try … catch` block

Handling multiple exceptions

```
try {
    setAge(age);
    openFile(filename);
}
catch(IllegalArgumentException e) {
    System.out.println("Unchecked exception!");
    System.err.println(e);
}
catch(IOException e) {
    System.out.println("Checked exception!");
    System.err.println(e);
}
finally {
    System.out.println("Print this anyways.");
}
```

You can use several `catch` blocks after the `try` block to catch different exceptions

Nested try … catch blocks

- It is possible to use *nested* `try…catch` blocks
  - Usually best to avoid and try to find another solution!

```
1   import java.util.*;
2   public class TryCatchTest2 {
3     public static void divide() {
4       Scanner input = new Scanner(System.in);
5       System.out.print("Give x: ");
6       int x = input.nextInt();
7       System.out.print("Give y: ");
8       int y = input.nextInt();
9       System.out.println("x / y = " + x/y);
10    }
```

```
11    public static void main(String[] args) {
12      try {
13        divide();
14      }
15      catch(Exception e1) {
16        System.out.println("y can't be zero!");
17        System.out.println("Try again.");
18        try {
19          divide();
20        }
21        catch(Exception e2) {
22          System.out.println("y still can't be zero!");
23          System.out.println("I give up.");
24        }
25      }
26    }
27  }
```

Nested `try…catch`

```
Give x: 6
Give y: 0
y can't be zero!
Try again.
Give x: 7
Give y: 0
y still can't be zero!
I give up.
```

Questions, comments?