1495 UNIVERSITY OF ABERDEEN

JC2002 Java Programming

Day 6: Exceptions (CS)

Tuesday, 7 October

1495 UNIVERSITY OF ABERDEEN

JC2002 Java Programming

Day 5, Session 1: Exception handling in Java

References and learning objects

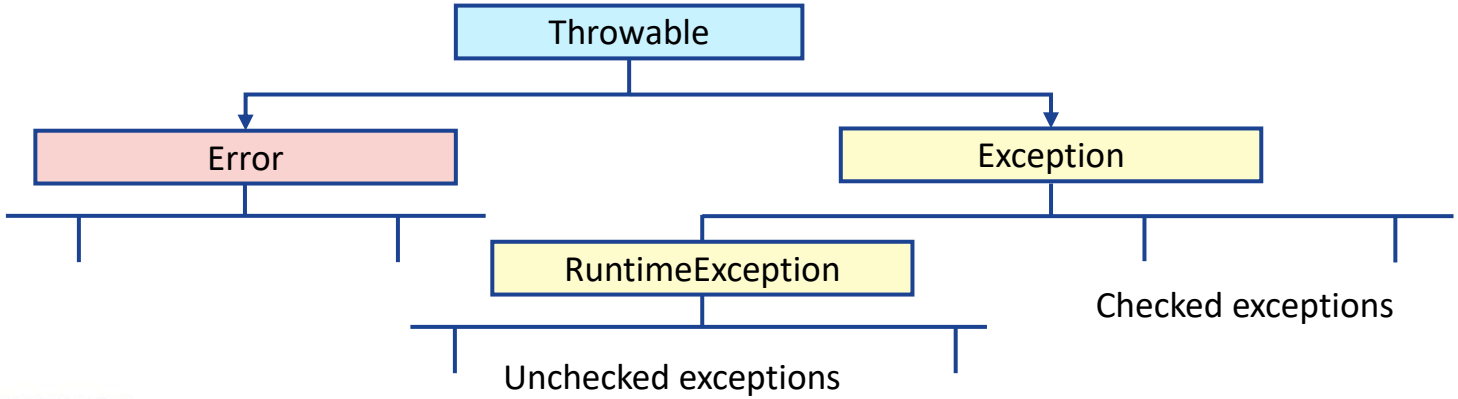
- Today's sessions are largely based on ***Java: How to Program***, Chapter 7, and ***Java in a Nutshell***
- After today's session, you should be able to:
  - Handle exceptions with `try…catch` structure in your Java code
  - Define and use your own custom exceptions

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
- ***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 runtime (out of memory, class not found etc.)



Error example (infinite recursion)

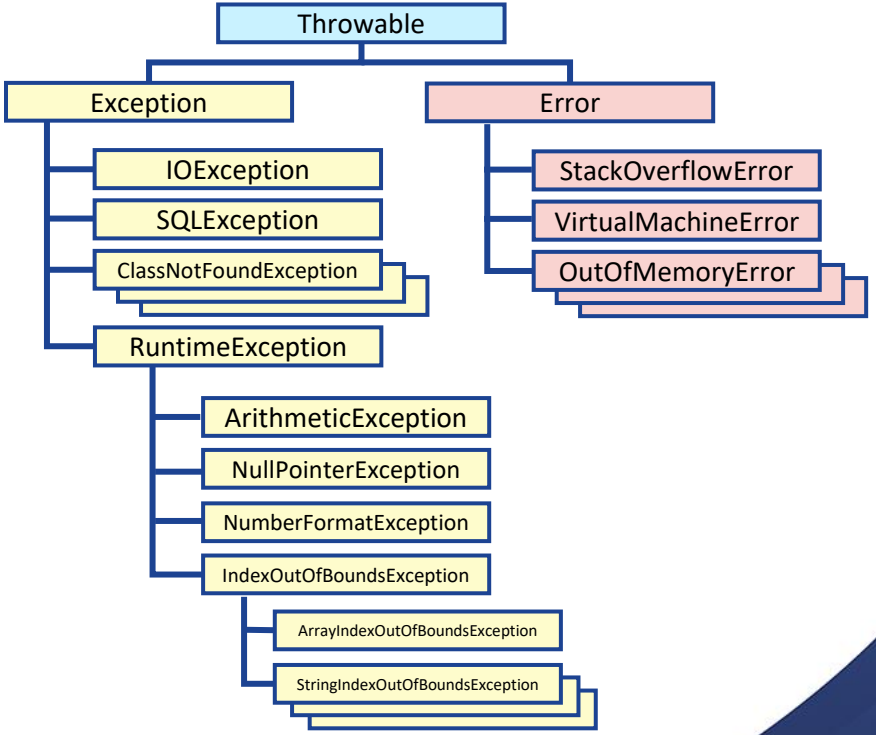
```java
public class SimpleRecursion2 {
    public static void recursiveLoop(int i, int max){
        System.out.print(i + " ");
        if(i < max) {
            recursiveLoop(i, max);
        }
    }
    public static void main(String[] args){
        recursiveLoop(1,10);
        System.out.println();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

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

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

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 exemptions* are all subclasses of **`Exception`**, except the **`RuntimeException`** subclasses that 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 **`throws`** keyword, otherwise the compiler will return an error

Checked exception example

```java
import java.io.*;
public class ExceptionTest1 {
    public static void main(String args[]) {
        FileInputStream inputStream = null;
        inputStream = new FileInputStream("file.txt");
        int m;
        while ((m = inputStream.read()) != -1) {
            System.out.print((char) m);
        }
        inputStream.close();
    }
}
```

`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:

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

This is only run if there was an exception

Program continues here, normally in any case

Example of try … catch

```java
import java.util.*;
public class TryCatchTest {
    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();
        try {
            System.out.println("x / y = " + x/y);
        }
        catch(Exception e) {
            System.out.println("y can't be zero!");
        }
    }
}
```

```
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!

```java
import java.util.*;
public class TryCatchTest2 {
  public static void divide() {
    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);
  }
```

```java
  public static void main(String[] args) {
    try {
      divide();
    }
    catch(Exception e1) {
      System.out.println("y can't be zero!");
      System.out.println("Try again.");
      try {
        divide();
      }
      catch(Exception e2) {
        System.out.println("y still can't be zero!");
        System.out.println("I give up.");
      }
    }
  }
}
```

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?

1495 UNIVERSITY OF ABERDEEN

JC2002 Java Programming

Day 6, Session 2: User-defined exceptions

Why user defined exceptions?

- The built-in exceptions cover almost all the general types of exceptions in programming
- However, in some cases custom exceptions can be beneficial:
  - To catch specific subsets of existing Java exceptions
  - To handle “business logic exceptions” not related to program errors, but e.g., data errors specific to the application
  - Custom exceptions allow handling at specific level of the program
- User defined exceptions can be created simply by inheriting from the existing exceptions

Example of user defined exception (1)

```java
import java.util.*;
class IntOverflowException extends Exception {
  public IntOverflowException(String str) {
      super(str);
  }
}
public class TestCustomException {
  static int fact(int x)
                throws IntOverflowException {
    int y=1;
    for(int i=1; i<=x; y *= i++) {
      if(i<x && (long)y*(long)i>Integer.MAX_VALUE) {
        throw new IntOverflowException("integer overflow");
      }
    }
    return y;
  }
  static int computeC(int n, int r)
                throws IntOverflowException {
    int res = fact(n)/(fact(r)*fact(n-r));
    return res;
  }
  public static void main(String args[])
  {
    try  {
      computeC(50,10); // compute C(n,k)
    }
    catch (IntOverflowException ex) {
      System.out.println(ex.getMessage());
    }
    System.out.println("continue...");
  }
}
```

Example of user defined exception (2)

```java
import java.util.*;
class IntOverflowException extends Exception {
  public IntOverflowException(String str) {
      super(str);
  }
}
public class TestCustomException {
  static int fact(int x)
                throws IntOverflowException {
    int y=1;
    for(int i=1; i<=x; y *= i++) {
      if(i<x && (long)y*(long)i>Integer.MAX_VALUE) {
        throw new IntOverflowException("integer overflow");
      }
    }
    return y;
  }
  static int computeC(int n, int r)
                throws IntOverflowException {
    int res = fact(n)/(fact(r)*fact(n-r));
    return res;
  }
  public static void main(String args[])
  {
    try  {
      computeC(50,10); // compute C(n,k)
    }
    catch (IntOverflowException ex) {
      System.out.println(ex.getMessage());
    }
    System.out.println("continue...");
  }
}
```

User defined exception. Note that constructor and call to super( ) is not obligatory, but it helps to implement the default functionality.

Example of user defined exception (3)

```java
import java.util.*;
class IntOverflowException extends Exception {
  public IntOverflowException(String str) {
      super(str);
  }
}
public class TestCustomException {
  static int fact(int x)
                throws IntOverflowException {
    int y=1;
    for(int i=1; i<=x; y *= i++) {
      if(i<x && (long)y*(long)i>Integer.MAX_VALUE) {
        throw new IntOverflowException("integer overflow");
      }
    }
    return y;
  }
  static int computeC(int n, int r)
                throws IntOverflowException {
    int res = fact(n)/(fact(r)*fact(n-r));
    return res;
  }
```

```java
  public static void main(String args[])
  {
    try  {
      computeC(50,10); // compute C(n,k)
    }
    catch (IntOverflowException ex) {
      System.out.println(ex.getMessage());
    }
    System.out.println("continue...");
  }
}
```

You need to use keyword `throws` to indicate which methods could throw the custom exception. Alternatively, you can inherit your exception from `RuntimeException`.

Example of user defined exception (4)

```java
import java.util.*;
class IntOverflowException extends Exception {
  public IntOverflowException(String str) {
      super(str);
  }
}
public class TestCustomException {
  static int fact(int x)
                throws IntOverflowException {
    int y=1;
    for(int i=1; i<=x; y *= i++) {
      if(i<x && (long)y*(long)i>Integer.MAX_VALUE) {
        throw new IntOverflowException("integer overflow");
      }
    }
    return y;
  }
  static int computeC(int n, int r)
                throws IntOverflowException {
    int res = fact(n)/(fact(r)*fact(n-r));
    return res;
  }
```

```java
  public static void main(String args[])
  {
    try  {
      computeC(50,10); // compute C(n,k)
    }
    catch (IntOverflowException ex) {
      System.out.println(ex.getMessage());
    }
    System.out.println("continue...");
  }
}
```

Exception is thrown if variable y (int) will overflow in the next round.

Example of user defined exception (5)

```java
import java.util.*;
class IntOverflowException extends Exception {
  public IntOverflowException(String str) {
      super(str);
  }
}
public class TestCustomException {
  static int fact(int x)
                throws IntOverflowException {
    int y=1;
    for(int i=1; i<=x; y *= i++) {
      if(i<x && (long)y*(long)i>Integer.MAX_VALUE) {
        throw new IntOverflowException("integer overflow");
      }
    }
    return y;
  }
  static int computeC(int n, int r)
                throws IntOverflowException {
    int res = fact(n)/(fact(r)*fact(n-r));
    return res;
  }
```

```java
  public static void main(String args[])
  {
    try  {
      computeC(50,10); // compute C(n,k)
    }
    catch (IntOverflowException ex) {
      System.out.println(ex.getMessage());
    }
    System.out.println("continue...");
  }
}
```

Our `try…catch` block. From experience, we know that computing $C(50,10)$ will cause `int` overflow.

Example of user defined exception (6)

```java
import java.util.*;
class IntOverflowException extends Exception {
  public IntOverflowException(String str) {
      super(str);
  }
}
public class TestCustomException {
  static int fact(int x)
                throws IntOverflowException {
    int y=1;
    for(int i=1; i<=x; y *= i++) {
      if(i<x && (long)y*(long)i>Integer.MAX_VALUE) {
        throw new IntOverflowException("integer overflow");
      }
    }
    return y;
  }
  static int computeC(int n, int r)
                throws IntOverflowException {
    int res = fact(n)/(fact(r)*fact(n-r));
    return res;
  }
  public static void main(String args[])
  {
    try  {
      computeC(50,10);
    }
    catch (IntOverflowException ex) {
      System.out.println(ex.getMessage());
    }
    System.out.println("continue...");
  }
}
```

```
$ java TestCustomException
integer overflow
continue...
$
```

Custom unchecked exception example

```java
import java.util.*;
class IntOverflowException extends RuntimeException {
}
public class TestCustomException2 {
  static int fact(int x) {
    int y=1;
    for(int i=1; i<=x; y *= i++) {
      if(i<x && (long)y*(long)i>Integer.MAX_VALUE) {
        throw new IntOverflowException();
      }
    }
    return y;
  }
  static int computeC(int n, int r) {
    int res = fact(n)/(fact(r)*fact(n-r));
    return res;
  }
  public static void main(String args[])
  {
    try  {
      computeC(50,10);
    }
    catch (IntOverflowException ex) {
      System.out.println("int overflow...");
    }
    System.out.println("continue...");
  }
}
```

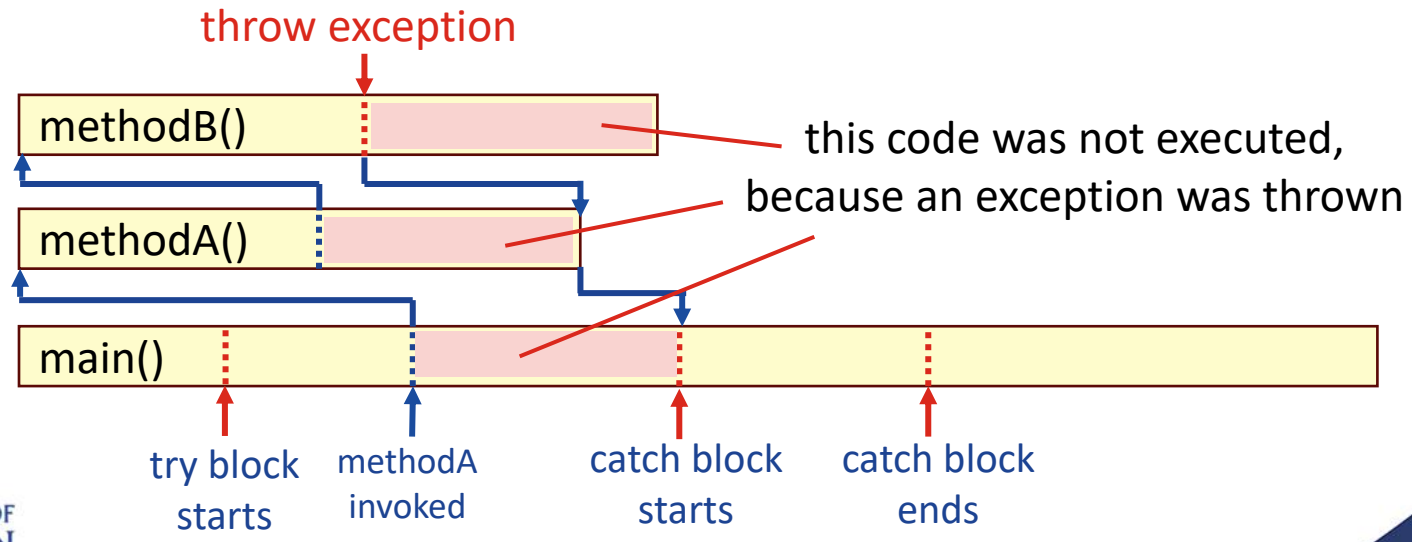
```
$ java TestCustomException2
int overflow
continue...
$
```

Simplified class without constructor inherited from `RuntimeException`

Keyword throws not required for unchecked exceptions

Code ignored due to an exception

- Note that when an exception is thrown, it is propagated through the call stack, until the exception is handled: some data may not be properly initialised!



Variables with no valid data assigned

```java
import java.util.*;
class IntOverflowException extends RuntimeException {
}
public class TestCustomException3 {
  static int fact(int x) {
    int y=1;
    for(int i=1; i<=x; y *= i++) {
      if(i<x && (long)y*(long)i>Integer.MAX_VALUE) {
        throw new IntOverflowException();
      }
    }
    return y;
  }
  static int computeC(int n, int r) {
    int res = fact(n)/(fact(r)*fact(n-r));
    return res;
  }
```

```java
  static int C;
  public static void main(String args[])
  {
    try  {
      C = computeC(10,5);
    }
    catch (IntOverflowException ex) {
      System.out.println("int overflow");
    }
    System.out.println("C = " + C);
  }
}
```

```
$ java TestCustomException3
int overflow
C = 0
$
```

Because an exception was thrown, `int C` does not have valid value!

Caveats of generic exception handlers

- A generic exception handler catching `Exception` superclass can give misleading information about the underlying problem
  - You should not expect that exceptions are always thrown for the same reason!
- Sometimes custom exceptions are thrown after the code catches a standard exception
  - You should provide a constructor that preserves the details of the error from the standard exception

Misinterpreted exception

```java
import java.util.*;
public class TryCatchTest2 {
  public static void divide() {
    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);
  }
 public static void main(String[] args) {
    try {
      divide();
    }
    catch(Exception e1) {
      System.out.println("y can't be zero!");
    }
  }
}
```

```
Give x: 5
Give y: abc
y can't be zero!
```

In this case, input is not numeric, and the exception thrown is `InputMismatchException`, not `ArithmeticException`

Custom exception example with cause (1)

```java
import java.util.*;
class DivisionException extends Exception {
  public DivisionException(String msg,
                           Throwable cause) {
    super(msg + cause.toString());
  }
}
public class TestCustomException4 {
  public static void divide()
            throws DivisionException {
    try {
      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);
    }
    catch(Exception e) {
      throw new DivisionException("division() failed due to ", e);
    }
  }
```

```java
  public static void main(String[] args) {
    try {
      divide();
    }
    catch(DivisionException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

Custom exception example with cause (2)

```
1   import java.util.*;
2   class DivisionException extends Exception {
3     public DivisionException(String msg,
4                              Throwable cause) {
5       super(msg + cause.toString());
6     }
7   }
8   public class TestCustomException4 {
9     public static void divide()
10              throws DivisionException {
11      try {
12        Scanner inpu
13        System.out.p
14        System.out.p
15        System.out.p
16      }
17      catch(Exceptio
18        throw new Di
19      }
20    }
```

```
21    public static void main(String[] args) {
22      try {
23        divide();
24      }
25      catch(DivisionException e) {
26        System.out.println(e.getMessage());
27      }
28    }
29  }
```

Define constructor that preserves the cause of the exception (original general exception catched)

Custom exception example with cause (3)

Catch the general exception and throw the custom (business) exception

```
1   import java.util.*;
                                  ception {
                                  msg,
                                  le cause) {
8   public class TestCustomException4 {
9     public static void divide()
10              throws DivisionException {
11      try {
12        Scanner input = new Scanner(System.in);
13        System.out.print("Give x: "); int x = input.nextInt();
14        System.out.print("Give y: "); int y = input.nextInt();
15        System.out.println("x / y = " + x/y);
16      }
17      catch(Exception e) {
18        throw new DivisionException("division() failed due to ", e);
19      }
20    }
```

```
21    public static void main(String[] args) {
22      try {
23        divide();
24      }
25      catch(DivisionException e) {
26        System.out.println(e.getMessage());
27      }
28    }
29  }
```

Custom exception example with cause (4)

```
1   import java.util.*;
2   class DivisionException extends Exception {
3     public DivisionException(String msg,
4                              Throwable cause) {
5       super(msg + cause.toString());
6     }
7   }
8   public class TestCustomException4 {
9     public static void divide()
10              throws DivisionException {
11      try {
12        Scanner input = new Scanner(System.in);
13        System.out.print("Give x: "); int x = input.r
14        System.out.print("Give y: "); int y = input.r
15        System.out.println("x / y = " + x/y);
16      }
17      catch(Exception e) {
18        throw new DivisionException("division() failed due to ", e);
19      }
20    }
```

```
21    public static void main(String[] args) {
22      try {
23        divide();
24      }
25      catch(DivisionException e) {
26        System.out.println(e.getMessage());
27      }
28    }
29  }
```

Catch the custom (business) exception and print out the underlying cause exception

Custom exception example with cause (5)

```java
import java.util.*;
class DivisionException extends Exception {
  public DivisionException(String msg,
                           Throwable cause) {
    super(msg + cause.toString());
  }
}
public class TestCustomException4 {
  public static void divide()
            throws DivisionException {
    try {
      Scanner input = new Scanner(System.in);
      System.out.print("Give x: "); int x = input.n
      System.out.print("Give y: "); int y = input.n
      System.out.println("x / y = " + x/y);
    }
    catch(Exception e) {
      throw new DivisionException("division() faile
    }
  }
```

```java
  public static void main(String[] args) {
    try {
      divide();
    }
    catch(DivisionException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

```
$ java CustomExceptionTest4
Give x: 56
Give y: 0
Division failed due to
java.lang.ArithmeticException: / by zero
$ java CustomExceptionTest4
Give x: abc
Division failed due to
java.util.InputMismatchException
```

Summary

- In Java, errors and other exceptional situations throw exceptions
  - Exceptions can be handled by `try…catch` structure
  - Checked exceptions must be either declared by keyword `throws` or handled in an exception handler,
  - Unchecked exceptions are usually caused by programming errors and do not need to be handled (instead the code should be fixed!)
- User defined (custom) exceptions can also be defined and handled
  - Useful to implement different exception handling procedures in the same part of the code

Questions, comments?