



# Java Programming

## Unit 7

Error Handling. Exceptions.

# Runtime errors

An exception is an run-time error that may stop the execution of your program. For example:

- someone deleted a file that a program usually reads
- The client calls the server, which doesn't respond
- A program variable should point at the instance of an object, but it was never instantiated.

Java compiler often forces you to add *exception handling* in your programs.

# Stack Trace

Run this program and you'll see a *stack trace* output in the system console.

```
1 class TestStackTrace{
2     TestStackTrace()
3     {
4         divideByZero();
5     }
6
7     int divideByZero()
8     {
9         // Purposely dividing by zero
9         return 25/0;
10    }
11    public
12    static void main(String[] args)
13    {
14        new TestStackTrace();
15
16    }
17 }
```

java TestStackTrace

Exception in thread "main"

java.lang.ArithmeticException: / by zero

at

TestStackTrace.divideByZero(TestStackTrace.java:9)

at TestStackTrace.<init>(TestStackTrace.java:4)

at TestStackTrace.main(TestStackTrace.java:14)



Read it from the bottom up!

# Walkthrough 1

- Download the source code from the Lesson 13 and import it into Eclipse
- Run the [TestStackTrace](#) program and observe the stack trace in the Console view
- Examine the code of the [TestStackTrace2](#), which handles this exception. Run this program

# Try-Catch Blocks

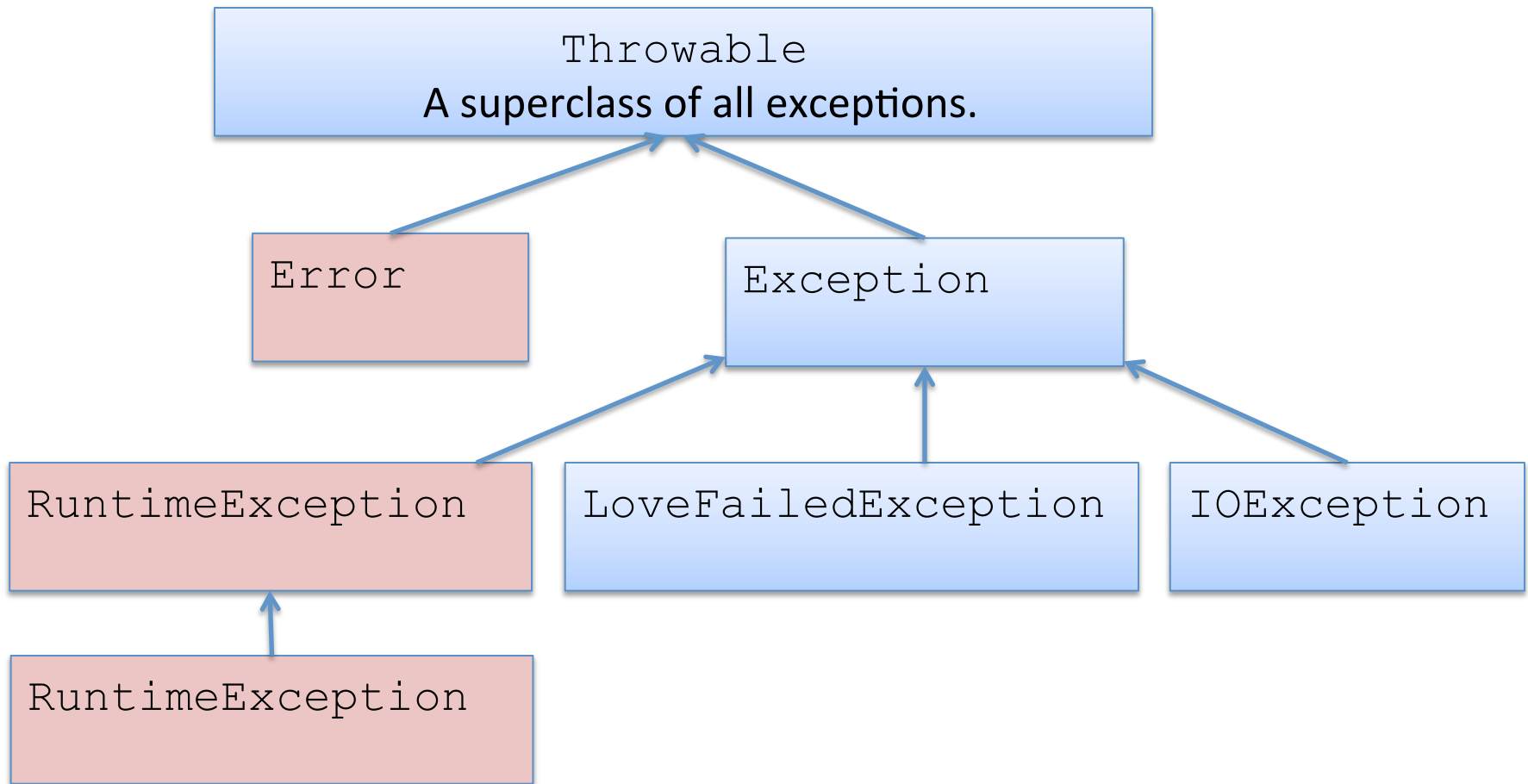
If a piece of code may result in a *run-time* error, Java *can* force you to enclose it in the `try-catch` block.

For example, reading a file may generate an `IOException`:

```
try{
    fileCustomer.read();    // the file may be corrupted or missing
}
catch (IOException e){
    System.out.println("There seems to be a problem with the file customers.");
}
```

*Checked exceptions* are the ones that you can foresee, but can't fix inside the program. For example if the file with the customer data is corrupted, there is nothing your program can do about it.

# Exceptions Hierarchy



Subclasses of the class **Error** and **RuntimeException** are fatal errors and are called *unchecked exceptions*, and are not required to be caught. Subclasses of **Exception** (excluding **RuntimeException**) are called *checked exceptions* and have to be handled in your code.

# Catching Multiple Exceptions Before Java 7

```
public void getCustomers(){
    try{
        fileCustomers.read(); // may throw an error

    } catch(FileNotFoundException e){
        System.out.println("Can not find file Customers");
    }catch(EOFException e1){
        System.out.println("Done with file read");
    }catch(IOException e2){
        System.out.println("Problem reading file " +
                           e2.getMessage());
    } catch (Exception e3){
        System.out.println("Something went wrong");
    }
}
```

# Catching Multiple Exceptions in Java 7

Java 7 introduced catching multiple exceptions in one catch block:

```
public void getCustomers(){
    try{
        fileCustomers.read(); // may throw an error

    } catch(FileNotFoundException | EOFException | IOException e){
        System.out.println("Problem reading file " +e.getMessage());
    } catch (Exception e1){
        System.out.println("Something else went wrong");
    }
}
```



# The **throws** Keyword

```
class CustomerList{

    void getAllCustomers() throws IOException{

        // Some other code goes here
        // Don't use try/catch if you are not handling exceptions
        // in this method
        file.read();
    }

    public static void main(String[] args){
        System.out.println("Customer List");

        // Some other code goes here

        try{
            // Since getAllCustomers() declares an exception,
            // either handle it over here, or re-throw it

            getAllCustomers();
        }
        catch(IOException e){
            // Exception is considered handled
            System.out.println("Customer List is not available");
        }
    }
}
```

## Propagation of Exceptions

If you are not planning to handle exceptions in a method, add a **throws** in the method declaration.

## Handling Exceptions

Now the caller of **getAllCustomers()** will have to handle the exception.

# Walkthrough 2

1. Visit the help page for the class `FileInputStream`.

<http://download.oracle.com/javase/7/docs/api/java/io/FileInputStream.html>

2. Browse the help info on various methods of this class and pay attention to their `throws` statements.


3. Visit the help page for the class `IOException` and look at its direct known subclasses:

<http://download.oracle.com/javase/7/docs/api/java/io/IOException.html>

Take a look at its direct known subclasses.

# The `finally` keyword

If there is a piece of code that must be executed *regardless* of the success or failure of the code inside `try{}`, put it under the clause `finally`.



```
try{
    file.read();
    //file.close(); don't close files inside try
}
catch(Exception e){
    e.printStackTrace();
}
finally{
    if (file != null){
        try{
            file.close();
        }catch(IOException e1){
            e1.printStackTrace();
        }
    }
}
```

Prior to Java 7, `finally` was the only right place to external resources like database connections or open files.

# Java 7: try statement with resources

*try-with-resources* that supports *automatic closing* of the resources without the need to use the **finally** clause:

```
InputStream myFileInputStream = null;

try (myFileInputStream=new FileInputStream("customers.txt");) {
    // the code that reads data from customers.txt goes here
} catch (Exception e){
    e.printStackTrace();
}
```

This works only if the resource implements **java.lang.AutoCloseable** or **java.io.Closeable** interface, which declares just one method **close()**.

See the example of **try** with automatic resource management is here:  
<http://java.dzone.com/articles/java-7-new-try-resources>

# The throw keyword

**throws** just declares, but **throw** actually throws the object.

```
class CustomerList{

    void getAllCustomers() throws Exception{

        // some other code goes here

        try{
            file.read(); // this line may throw an exception
        } catch (IOException e) {

            // Log this error here, and re-throw another exception
            throw new Exception ("Customer List is not available "+
                                e.getMessage());

        }
    }

    public static void main(String[] args){
        System.out.println("Customer List");
        ...
        try{

            // Since the getAllCustomers() declares an exception,
            // you have to either handle or re-throw it
            getAllCustomers();

        }
        catch(Exception e){
            System.out.println(e.getMessage());
        }
    }
}
```

# Defining & Throwing Custom Exceptions

```
class TooManyBikesException extends Exception{  
    TooManyBikesException (String msgText){  
        super(msgText);  
    }  
}
```

```
class BikeOrder{  
    ...  
    static void validateOrder(String bikeModel,  
        int quantity) throws TooManyBikesException{  
  
        // perform some data validation, and if the entered  
        // the quantity or model is invalid, do the following:  
  
        throw new TooManyBikesException("Can not ship" +  
            quantity + " bikes of the model " + bikeModel);  
    }  
}
```

```
class OrderWindow extends JFrame{  
    ...  
    void actionPerformed(ActionEvent e){  
  
        // the user clicked on the "Validate Order" button  
  
        try{  
            bikeOrder.validateOrder("Model-123", 50);  
  
            // the next line will be skipped in case of exception  
            txtResult.setText("Order is valid");  
  
        } catch(TooManyBikesException e){  
  
            txtResult.setText(e.getMessage());  
  
        }  
    }  
}
```

# Adding useful info to Exceptions

```
class TooManyBikesException extends Exception{

    // Declare an application-specific property
    ShippingErrorInfo shippingErrorInfo;

    TooManyBikesException (String msgText,
                           ShippingErrorInfo shippingErrorInfo){

        super(msgText);
        this.shippingErrorInfo = shippingErrorInfo;
    }
}
```

1. Create an instance of TooManyBikesException providing the error details in the ShippingErrorInfo.
2. Throw TooManyBikesException.
3. In the code that handles TooManyBikesException extract the shippingErrorInfo object from the exception object and display it to the user and/or log the error.

# Java 7: More Inclusive Type Checking

```
1 import java.io.IOException;
2 import java.sql.SQLException;
3
4 public class MultyRethrow {
5
6     public static void main(String[] args) {
7
8         try{
9             readSomeData();|
10        } catch(IOException | SQLException e){
11            e.printStackTrace();
12        }
13    }
14
15    static void readSomeData() throws IOException, SQLException{
16        try{
17            // The code that may generate IOException or SQLException
18            // goes here
19            SQLException esql = new SQLException("Table Cust doesn't exist");
20            throw esql;
21
22        } catch(Exception e){
23            // rethrow the exception
24            throw e;
25        }
26    }
27 }
```



# Java 7: More Inclusive Type Checking

```
1 import java.io.IOException;
2 import java.sql.SQLException;
3
4 public class MultyRethrow {
5
6     public static void main(String[] args) {
7
8         try{
9             readSomeData();|
10        } catch(IOException | SQLException e){
11            e.printStackTrace();
12        }
13    }
14
15    static void readSomeData() throws IOException, SQLException{
16        try{
17            // The code that may generate IOException or SQLException
18            // goes here
19            SQLException esql = new SQLException("Table Cust doesn't exist");
20            throw esql;
21
22        } catch(Exception e){
23            // rethrow the exception
24            throw e;
25        }
26    }
27 }
```

---

java.sql.SQLException: Table Cust doesn't exist  
at MultyRethrow.readSomeData(MultyRethrow.java:20)  
at MultyRethrow.main(MultyRethrow.java:9)

# Java 7: Final Rethrow

The `final` keyword with exceptions means that you can't change the exception object `e`:

```
try{
    // some code goes here
} catch (final Throwable e){
    // some exception handling code goes here
    throw e;
}
```

If a catch block handles more than one exception, the catch parameter is implicitly final. Can't `ex` modify below:

```
catch (final IOException | SQLException ex) {
    logger.log(ex);
    throw ex;
}
```

# Homework

- Complete the assignments from the Try It sections for Lessons 13.
- You're selling bikes online and have one truck to deliver bikes. Create a Swing app, where the user can select a bicycle model and the quantity. If requested quantity won't fit in your truck, throw the `TooManyBikesException` and display the meaningful message to the user.

# Additional Reading

Study Oracle's tutorial on exceptions:

<http://bit.ly/1nO3wIO>

Read about multi-catch exceptions: <http://bit.ly/N7NMTP>

Read about using Java 7 `AutoCloseable` interface:

<http://bit.ly/1cYXwMi>

Watch the presentation about the Java Garbage Collector:

<http://bit.ly/18TBK7K>