

# Test Automation

## Lecture 8 –

### Exceptions & Generics



IT Learning &  
Outsourcing Center

Lector: Milen Strahinski  
Skype: strahinski  
E-mail: [milen.strahinski@pragmatic.bg](mailto:milen.strahinski@pragmatic.bg)  
Facebook: <http://www.facebook.com/LamerMan>  
LinkedIn: <http://www.linkedin.com/pub/milen-strahinski/a/553/615>

[www.pragmatic.bg](http://www.pragmatic.bg)

Copyright © Pragmatic LLC



# Summary

- Exception
  - What's an exception
  - Types of exceptions
  - Handling Exceptions
  - Custom Exceptions and throwing exceptions
  
- Generics



# What is an exception?

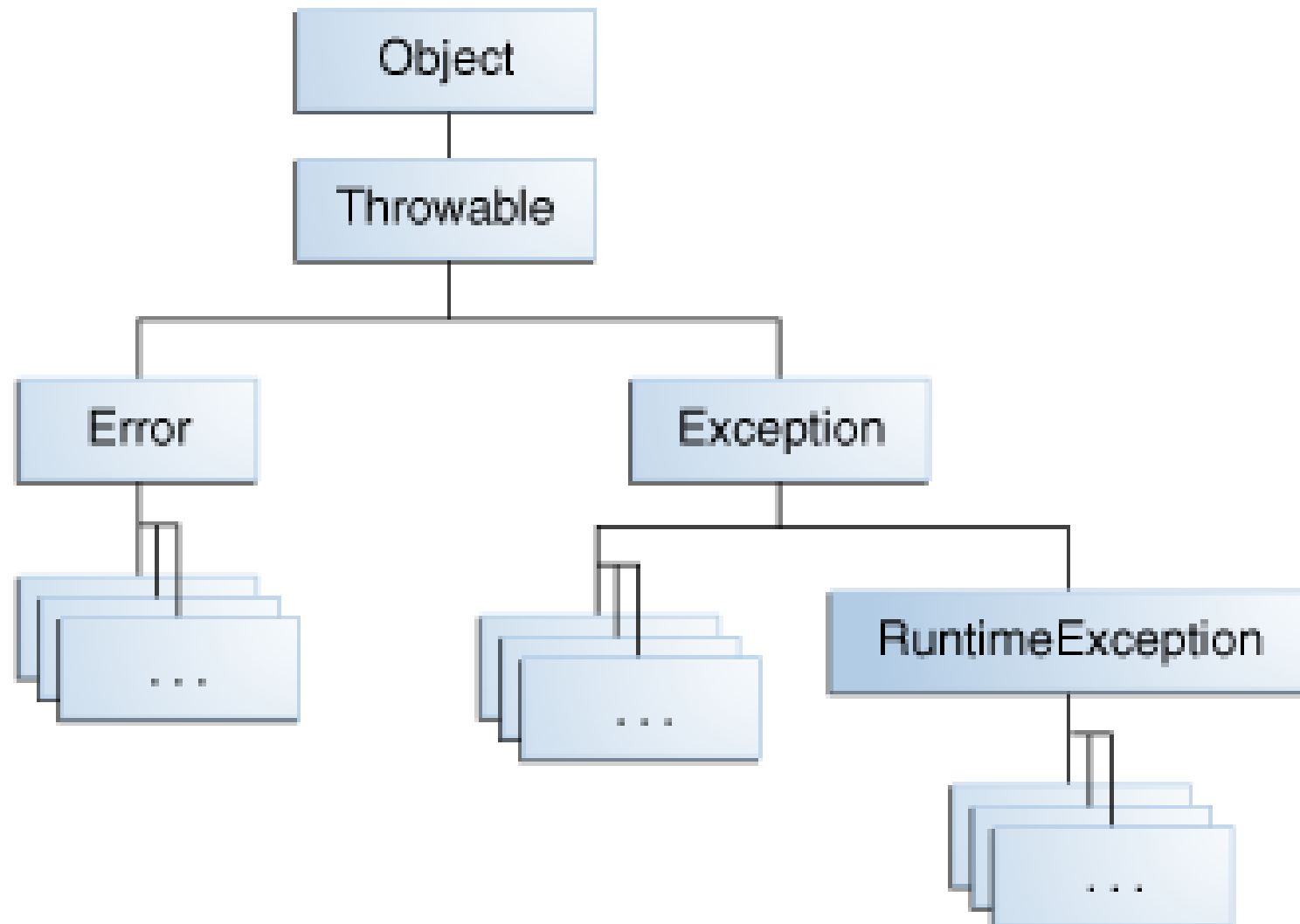
- The term *exception* is shorthand for the phrase "exceptional event."
- When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an *exception object*, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called *throwing an exception*.



- Ok but what to do when there is an exception?
- Usually if there is an exception the program will CRASH. However what can be done is to catch this exceptions and do something when they appear without CRASHING the whole program



# Exception Fundamentals





- Throwable - All exception types are subclasses of this class
- Exception(**checked exceptions**) - This class represents an exceptional condition that user programs should catch. This is also the class that needs to be subclassed to create a new custom exception
- RuntimeException - is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine. RuntimeException and its subclasses are *unchecked exceptions*
- Error – System related exception. Shouldn't be handled directly by the code



# Uncaught Exception

```
public static void main(String[] args) {  
    int a = 5;  
    int b = 0;  
  
    System.out.println(a / b);  
}
```



# Uncaught Exception

The screenshot shows an IDE's Error Log window with the following tabs: Problems, Javadoc, Declaration, Console, and Error Log. The Error Log contains the following text:

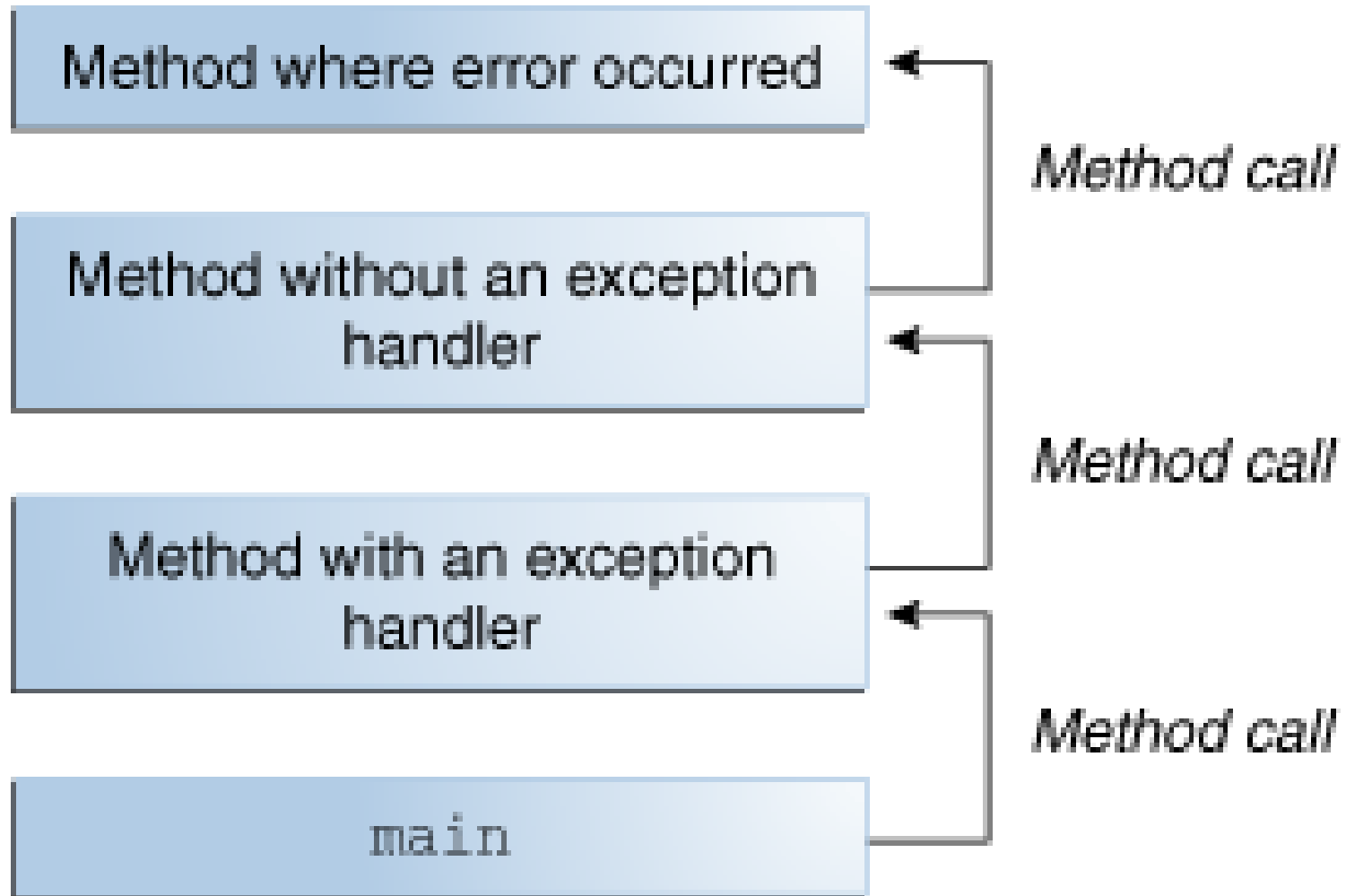
```
<terminated> UncaughtException [Java Application] C:\Program Files\Java\jdk1.7.0_17\bin\javaw.exe (Oct 11, 2013 7:17:48 AM)  
Exception in thread "main" java.lang.ArithmeticException: / by zero  
    at demo.UncaughtException.main(UncaughtException.java:9)
```

- When an exception occurs, the normal flow of the program is terminated.
- Exceptions must be immediately dealt with
- Execution continues to the first available exception handler capable of handling the exception that just occurred





# What is StrackTrace





# Exception Catching

## General Form

```
try {  
    this.throwNewCustomException();  
    this.throwNewIllegalArgumentException();  
    this.throwNewIOException();  
} catch (CustomException e) {  
    // handle exception here  
} catch (IOException e) {  
    // handle exception here  
}finally{  
    System.out.println("This is always performed");  
}
```



# Chained Exceptions

- An application often responds to an exception by throwing another exception.
- In effect, the first exception causes the second exception.
- It can be very helpful to know when one exception causes another.
- Chained Exceptions help the programmer do this.



# Chained Exceptions example

- In this example, when an IOException is caught, a new SampleException exception is created with the original cause attached and the chain of exceptions is thrown up to the next higher level exception handler.

```
try {  
    //...  
} catch (IOException e) {  
    throw new SampleException("Other IOException", e);  
}
```



# More about Throwable class

An instance of Throwable class contains

- Message
- Stacktrace
- Cause (instance of Throwable)



# More about Throwable class

## ■ Constructors:

```
public Throwable()  
public Throwable(String message)  
public Throwable(Throwable cause)  
public Throwable(String message, Throwable cause)
```

## ■ Important methods:

```
public String getMessage()  
public Throwable getCause()  
public void printStackTrace()  
public StackTraceElement[] getStackTrace()
```



# Exception Chaining

```
package other;

public class TestChainedException {
    public static void main(String[] args) {
        String s = null;
        testMethod(s);
    }

    public static void testMethod(String s) {
        try {
            System.out.println(s.length());
        } catch (NullPointerException npe) {
            throw new RuntimeException("Error when trying
to print the string's length", npe);
        }
    }
}
```



# How exceptions should be shown to the end user

- The end user is not programmer
- So, it's not a good practice to show technical details (stacktrace) to the end user
- Instead, nice message should be shown
- If we want, we can add technical information but it should be shown only if the user want to see it





# Question

- What happens with this code?

```
try {  
    throw new IOException();  
} catch (Exception e) {  
    e.printStackTrace();  
} catch (IOException e) {  
    e.printStackTrace();  
}
```



# Answer

## Compilation error!

- Unreachable catch block for IOException. It is already handled by the catch block for Exception
- Because IOException extends class Exception, so the second catch block will never execute.



# Tips

- We can handle multiple exceptions using catch block with parent class.
- This is useful when we want to handle more than one exceptions in the same way (we use a (too) general exception handler)

# Log info & re-throwing exception



- Sometimes we want to handle the exception just for a moment, use it for something (write in the log) and then re-throw it, because we can't handle it at all.
- Keyword *throw* is used (we saw it in the previous slides)

```
try {  
    //...  
} catch (IOException e) {  
    System.out.println("Error in testMethod: " + e.getMessage());  
    throw new IOException("Other IOException", e);  
}
```



# Defining own exceptions

- Just extends the class Exception
- Do not create a subclass of RuntimeException or **throw a RuntimeException**
- If we need, we can add some fields to these which is inherited by Exception
- It's good practice each module to throw only his own exceptions
- For readable code, it's good practice to append the string Exception to the names of all classes that inherit from the Exception class.



# Defining own exceptions

```
public class CustomException extends Exception{  
  
    public CustomException () {  
        super();  
    }  
    public CustomException (String message, Throwable cause) {  
        super(message, cause);  
    }  
    public CustomException (String message) {  
        super(message);  
    }  
    public CustomException (Throwable cause) {  
        super(cause);  
    }  
}
```



# Finally block

- The finally block always executes when the try block exits.
- This ensures that the finally block is executed even if an unexpected exception occurs
- it allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue, or break.
- Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated.



# Finally block

The finally block is a key tool for preventing resource leaks. When closing a file or otherwise recovering resources, place the code in a finally block to ensure that resource is always recovered.

```
try {  
    // some code which open PrintWriter out  
} catch (Exception e) {  
    //.. handle exception  
} finally {  
    if (out != null) {  
        System.out.println("Closing PrintWriter");  
        out.close();  
    } else {  
        System.out.println("PrintWriter not open");  
    }  
}
```





# Try-with-Resource

- Try-with-resource is a new java 7 feature created mainly to ensure the correct disposal of the resource associated with the statement
- The try-with-resource ensures that each resource is closed at the end of the statement.
- Any object that implements the `java.lang.AutoClosable`, which in effect are all `java.io.Closable` instances (such as `java.util.Scanner`)



# Try-with-resource example

```
try (AutoClosable instance ) {  
  
}
```

- Never forget that a try-with-resource is still a try – catch like clause therefore the following code is viable

```
■ try (InnerResource resource = new InnerResource()) {  
    // some code heres  
} catch (Exception e) {  
    e.printStackTrace();  
}
```



# Multi-Catch block

After java 7 we also have

```
try {  
.....  
} catch(ClassNotFoundException | SQLException ex) {  
    ex.printStackTrace();  
}
```



# What is a Generic?

- A way to write code independent of a type





# Generics Basics

- At its core - the term *generics* means *parameterized types*
- Using generics, it is possible to create a single class, that automatically works with different types of data.  
*For example: Printer which has different type of Cartridges.*
- A class, interface, or method that operates on a parameterized type is called *generic*, as in *generic class* or *generic method*
- *Why not just use Class Object?*



# Generics Example

```
public class Generic<T> {  
  
    private T field;  
  
    public Generic(T field) {  
        this.field = field;  
    }  
  
    public T getField() {  
        return field;  
    }  
  
    public void setField(T field) {  
        this.field = field;  
    }  
  
    public void printGenericType(){  
        System.out.println("Generic Filed is of type " + field.getClass().getName());  
    }  
  
    public static void main(String[] args) {  
        Generic<String> str = new Generic<String>("This is my field");  
        str.printGenericType();  
  
        Generic<Number> num = new Generic<Number>(50);  
        num.printGenericType();  
    }  
}
```



# Generic Levels

- **Class level**
  - Specify a type as a variable
  - Class of Type T
- **Method level**
  - Specify a type of a parameter
  - Parameter of Type T





# Syntax

- **Generic Class** - accessor `class ClassName<GenericDeclaration>`

```
public class Generic<T>
```

- **Generic Method** - accessor `<GenericDeclaration> returnType name()`

```
public <P> void showType(P arg)
```





# Bounded types

- Sometimes we wish to **restrict** the type of the generic. We want to tell the compiler some how that we want to pass in only a family of classes and **not** all possible types of classes.
- To do so we use the keyword: **extends**
- An example would be:

```
public class Printer<T extends ICartridge> {  
}
```



# Erasure

- Java generics are erasure type
  - meaning that they exist only at compile-time after which they are “erased”

Erasure types were created to preserve backward compatibility of the java bytecode(the generics does not exist in the bytecode, only compile-time). Old non generic programs need to be able to run on new modern virtual machines.



# Generic Restrictions

- Type Parameters **cannot** be instantiated – meaning:

```
T someReference = new T(); //compile error
```

- **static** fields **cannot** use a type parameter (i.e. T) declared by the enclosing class – meaning:

```
private static T someReference; //compile error
```

- You can **still** create generic static methods, but they must specify their own generic type

- You **cannot** instantiate an array of elements from the generic type

```
Box<String>[] someReference = new Box<String>[5]; //compile error
```

- and some other restrictions under:

- <http://docs.oracle.com/javase/tutorial/java/generics/restrictions.html>