Test Automation
Lecture 6 –
Exceptions & Strings

IT Learning &
Outsourcing Center

Lector: Peter Manolov
Skype: nsGhost1
E-mail: p.manolov@gmail.com

www.pragmatic.bg

2013 – 2014

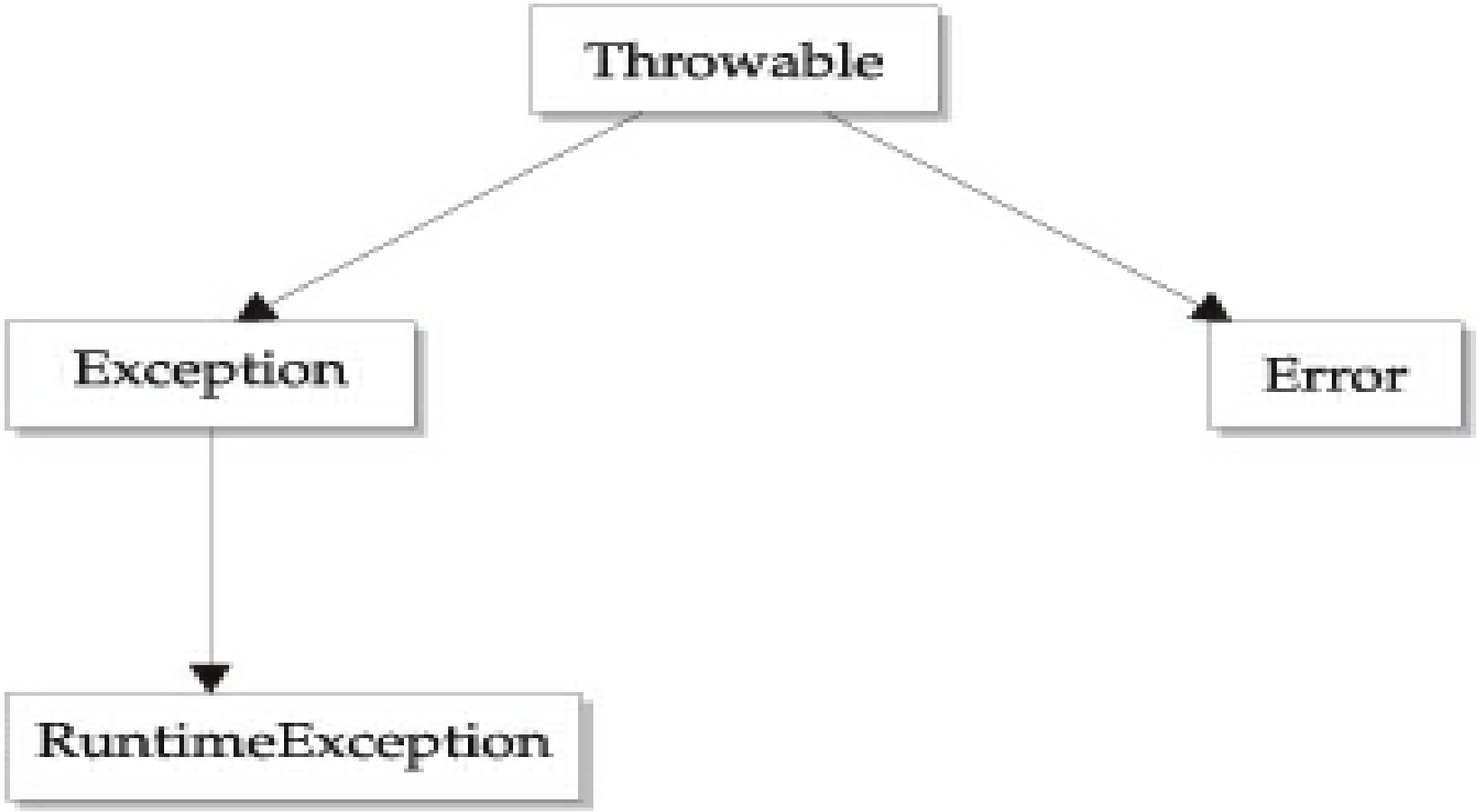
Summary

- Exception
  - What's an exception
  - Types of exceptions
  - Handling Exceptions
  - Custom Exceptions and throwing exceptions
- String
  - String Processing
  - StringBuilder

Exception Fundamentals- General Form


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

Exception Fundamentals



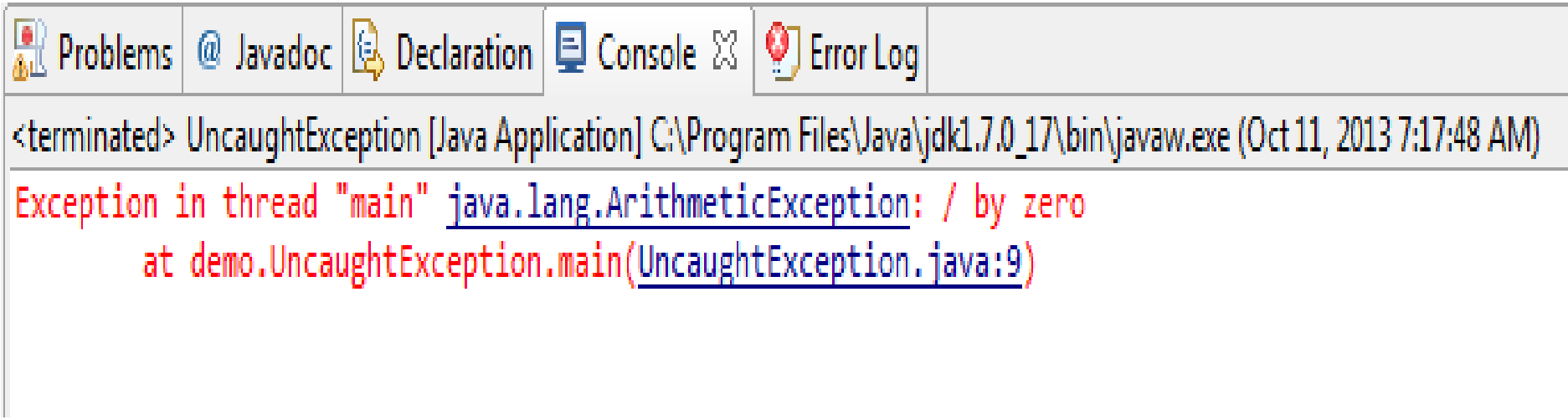
- Throwable - All exception types are subclasses of this class
- Exception - This class represent an exceptional conditions that user programs should catch. This is also the class that need to be subclassed to create a new custom exception
- RuntimeException - is the superclass of those exceptions that 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



- 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 occured

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 (instace 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 {
   //..
} catch (Exception e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

Answer

Compilaton 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)

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) {
 logger.log(Level.WARNING, "Error in testMethod: " + e.getMessage());
 throw new SampleException("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{

private static final long serialVersionUID = 905082714139195048 3L;

        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();
}
```

What Is a String?

- *Strings* are sequences of characters
  - Represented by the String class
- A *String* object holds a sequence of characters
- *String* objects are read-only (immutable)
  - Their values cannot be changed after creation (this is by design behavior)
- The String class represents all strings in Java

Creating a new string

- Using the string literal - double-quoted constant

```
String lastName = "John";
```

- Concatenate strings:

```
String fullName = firstName + " " + lastName;
```

- Use a constructor:

```
String fullName = new String("John Smith");
```

Concatenating Strings

- Use the + operator to concatenate strings

```
System.out.println("Name = " + name);
```

- You can concatenate primitives and strings

```
int age = getAge();
System.out.println("Age = " + age);
```

- String.concat() is another way to concatenate strings , it behaves the same ways as the + operator

```
System.out.println("Name = " .concat(name));
```

String Operations

- How to find the length of a string:
  - Use the length() method

```
String str = "John";
int len = str.length(); // len = 4
```

- How to find the character at a specific position:
  - Use the charAt(index) method
- Positions are counted from 0 to length()-1

```
String str = "John";
char c = str.charAt(1); // c = 'o'
```

String Operations

- How to extract a substring of a string:
  - Use the following method :

```
String substring(int beginIndex, int endIndex);
```

  - The symbol at the position endIndex is not part of the result!
- Example :

```
String s = "How are strings processed in Java?";
String substr = s.substring(8,15); // strings
```

String Operations

- How to find the index of a substring

```
int indexOf(String str);
int lastIndexOf(String str);
```

- Examples:

```
String str = "Java is the best language ever!";
System.out.println("\"" + str+ "\".indexOf(Java)= " + str.indexOf("Java"));
System.out.println("\"" + str+ "\".indexOf(best)= " + str.indexOf("best"));
System.out.println("\"" + str+ "\".indexOf(BEST)= " + str.indexOf("BEST"));
// IndexOf is case sensetive. -1 means not found
System.out.println("\"" + str+ "\".indexOf(gua)= " + str.indexOf("gua"));
```

Comparing Strings

- Use equals() to perform case-sensitive compare

```
String passwd = connection.getPassword();
if (passwd.equals("fgHPUw"))…  // Case is important
```

- Use equalsIgnoreCase()if you want to ignore the case:

```
String cat = getCategory();
if (cat.equalsIgnoreCase("Movie"))  …
// We just want the word to match
```

- The == operator compares the references of the String objects
- The .equals(…) method compares the contents of the strings
- This example shows the difference:

```
String text1 = new String("some string");
String text2 = new String("some string");
boolean incorrectCompare = (text1 == text2); // false
boolean correctCompare = (text1.equals(text2)); // true
```

Empty and null Strings

- The String objects can have a value of **null**
  - Remember strings are Objects not primitives

```
String text = null;
```

- The empty string in not a **null** string

```
String empty = "";
```

- Calling methods of a **null** string causes *NullPointerExceptipon*

```
String s = null;
String empty = "";
boolean equal = s.equals(empty);
// NullPointerExceptipon will be thrown
```

Splitting Strings

- Use the method:

```
String[] split(String regularExpression)
```

Example:

```
String listOfBeers = "Amstel, Zagorka, Tuborg, Becks.";
String[] beers = listOfBeers.split("[, .]+");
System.out.println("Available  beers are:");
for (String beer : beers) {
System.out.printf(" - %s\n", beer);
}
```

- *You need to have a basic understanding of regular expressions*

Object.toString()

- Use the Object.toString() to convert the current instance into string
- Your class can override toString()
- System.out.println() automatically calls an object's toString() method when a reference is passed to it

String.valueOf()

- Use String.valueOf(): to convert a primitive to a string

```
String seven = String.valueOf(7);
```

Constructing Strings

- **Strings are immutable**
  - concat(), replace(), trim(), … return new string, do not modify the old one
- **Do not use "+" for strings in a loop!**
  - **It runs very inefficiently!**

```
public String countChars(char ch, int count) {
String result = "";
for (int i=0; i<count; i++)
   result += ch;
   return result;
}
```

StringBuilder

- Use the StringBuilder class for modifiable strings of characters:

```
public String reverseIt(String s) {
   StringBuilder sb = new StringBuilder();
   for (int i = s.length()-1; i >= 0; i--) {
      sb.append(s.charAt(i));
   }
   return sb.toString();
}
```

- Use StringBuilder if you need to keep adding characters to a string

StringBuilder methods

- StringBuilder(int capacity) constructor allocates in advance buffer memory of a given size
  - By default 16 characters are allocated
- capacity() returns the currently allocated space (in characters)
- length() returns the length of the string
- charAt(int index) returns the char value at given position
- setCharAt(int index, char ch) changes a single character

StringBuilder methods

- append(…)
  - appends string or other type after the last character in the buffer
- delete(int start, int end)
  - removes the characters in given range
- insert(int offset, String str)
  - inserts given string at given position
- replace(int start, int end, String str)
  - replaces a substring by a given string
- toString()
  - converts the StringBuilder to String object