PROGRAMMING III JAVA LANGUAGE

COURSE 5

PREVIOUS COURSE CONTENT

- **Collections**
  - Utilities classes
- **Comparing objects**
- **Lambda expressions**
- **Generics**
  - Wild Cards
  - Restrictions

COURSE CONTENT

- **Collections**
  - Streams
  - Aggregate operations

- **Exceptions**

COLLECTIONS

- **What is a collection in Java?**
  - Containers of Objects which by polymorphism can hold any class that derives from Object
  - GENERICS make containers aware of the type of objects they store
    - from Java 1.5

JAVA 8 STREAMS

- ❑**What are streams?**
  - ❑Streams are not related to `InputStreams`, `OutputStreams`, etc.
  - ❑Streams are NOT data structures but are wrappers around Collection that carry values from a source through a pipeline of operations.
  - ❑Stream represents a sequence of objects from a source, which supports aggregate operations

JAVA 8 STREAMS

- **Streams characteristics**
  - Sequence of elements − A stream provides a set of elements of specific type in a sequential manner. A stream gets/computes elements on demand. It never stores the elements.
  - Source − Stream takes Collections, Arrays, or I/O resources as input source.
  - Aggregate operations − Stream supports aggregate operations like filter, map, limit, reduce, find, match, and so on.
  - Pipelining − Most of the stream operations return stream itself so that their result can be pipelined. These operations are called intermediate operations and their function is to take input, process them, and return output to the target. `collect()` method is a terminal operation which is normally present at the end of the pipelining operation to mark the end of the stream.
  - Automatic iterations − Stream operations do the iterations internally over the source elements provided, in contrast to Collections where explicit iteration is required.

STREAMS

- **Stream types**
  - `stream()` – Returns a sequential stream considering collection as its source.
  - `parallelStream()` – Returns a parallel Stream considering collection as its source.

Example

```
List<String> strings =
    Arrays.asList("abc", "", "bc", "efg", "abcd","",
                  "jkl");
List<String> filtered =
    strings.stream()
           .filter(string -> !string.isEmpty())
           .collect(Collectors.toList());
```

CREATING STREAMS

- **From individual values**
  - `Stream.of(val1, val2, …)`

- **From array**
  - `Stream.of(someArray)`
  - `Arrays.stream(someArray)`

- **From List (and other Collections)**
  - `someList.stream()`
  - `someOtherCollection.stream()`

CREATING STREAMS

❑ **`Stream.builder()`**

```
Stream<String> streamBuilder =Stream.<String>builder()
         .add("a").add("b").add("c")
         .build();
```

❑ **`Stream.generate()`**

```
Stream<String> streamGenerated =
         Stream.generate(() -> "element").limit(10);
```

❑ **`Stream.iterate()`**

```
Stream<Integer> streamIterated =
         Stream.iterate(40, n -> n + 2).limit(20);
```

CREATING STREAMS

❑**Stream of Primitives**

```
IntStream intStream = IntStream.range(1, 3);
LongStream longStream = LongStream.rangeClosed(1, 3);
Random random = new Random();
DoubleStream doubleStream = random.doubles(3);
```

❑**Stream of *String***

```
IntStream streamOfChars = "abc".chars()
 Stream<String> streamOfString =
      Pattern.compile(", ").splitAsStream("a, b, c");
```

STREAM PIPELINE

- Perform a sequence of operations over the elements of the data source and aggregate their results

- Parts
  - source
  - intermediate operation(s)
    - return a new modified stream
    - can be chained
  - terminal operation
    - Only one terminal operation can be used per stream.
    - The result of a interrogation
    - Example
      - Predefined operation: `count(), max(), min(), sum()`

STREAM PIPELINE

Example

```
List<String>strings =
        Arrays.asList("abc", "", "bc", "efg",
                      "abcd","", "jkl");

//get count of empty string
int count = strings.stream()
             .filter(string -> string.isEmpty())
             .count();
```

ORDER OF THE OPERATIONS

```
List<String> list = Arrays.asList("one", "two", "three", "four");

long size = list.stream().map(element -> {
    System.out.println("Call map method");
    return element.substring(0, 3);
    }).skip(2).count();
System.out.println("size" + size);

size = list.stream().skip(2).map(element -> {
    System.out.println("Call map method");
    return element.substring(0, 3);
    }).count();
System.out.println("size" + size);
```

What is the result of the following code?

ADVANCED OPERATIONS

- **collect**
  - transform the elements of the stream into a different kind of result
- **reduce**
  - combines all elements of the stream into a single result

```
class Person {
  String name;
  int age;
  Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
  @Override
   public String toString() {
      return name;
  }
}
List<Person> persons =
   Arrays.asList( new Person("Max", 18),
               new Person("Peter", 23),
               new Person("Pamela", 23),
               new Person("David", 12));
```

ADVANCED OPERATIONS. COLLECT

```
List<Person> filtered = persons .stream()
           .filter(p -> p.name.startsWith("P"))
           .collect(Collectors.toList());
System.out.println(filtered);
```

```
Map<Integer, List<Person>> personsByAge = persons .stream()
        .collect(Collectors.groupingBy(p -> p.age));
personsByAge .forEach((age, p) -> System.out.format("age %s: %s\n", age, p));
```

collect

```
Double averageAge = persons .stream()
           .collect(Collectors.averagingInt(p -> p.age));
System.out.println(averageAge);
```

```
IntSummaryStatistics ageSummary = persons .stream()
             .collect(Collectors.summarizingInt(p -> p.age));
System.out.println(ageSummary);
```

reduce

ADVANCED OPERATIONS. COLLECT

Exercise

Transform the following collect operation from collection Map<Integer, List<Person> to collecting for each different age the number of persons having that age

```
Map<Integer, List<Person>> personsByAge = persons .stream()
       .collect(Collectors.groupingBy(p -> p.age));
personsByAge .forEach((age, p) ->
                             System.out.format("age %s: %s\n", age, p));
```

Solution

```
Map<Integer, Long> personsByAge = persons .stream()
       .collect(Collectors.groupingBy(p -> p.age, Collectors.counting()));
personsByAge .forEach((age, nr) ->
                              System.out.format("age %s: %s\n", age, nr));
```

ADVANCED OPERATIONS. REDUCE

❑ **find the oldest person**

```
persons
  .stream()
  .reduce((p1, p2) -> p1.age > p2.age ? p1 : p2)
  .ifPresent(System.out::println);
```

❑ **determine the sum of ages from all persons**

```
Integer ageSum = persons
         .stream()
         .reduce(0, (sum, p) -> sum += p.age,
                    (sum1, sum2) -> sum1 + sum2);
System.out.println(ageSum);
```

EXAMPLE

```
Person result = persons.
          .stream()
          .filter(x -> "michael".equals(x.getName()))
         . findAny()
          .orElse(null);

Person result = persons
           .stream()
           .filter(x -> { if("michael".equals(x.getName()) &&
              21==x.getAge()){ return true; } return false; })
           .findAny()
           .orElse(null);
```

ERRORS

- ❑ **What are errors?**

  - ❑ The state or condition of being wrong in conduct or judgement

  - ❑ A measure of the estimated difference between the observed or calculated value of a quantity and its true value

ERRORS

- ❑ **Errors Types**
  - ❑ Syntax errors
    - ❑ Arise because the rules of the language have not been followed. They are detected by the compiler.
  - ❑ Runtime errors
    - ❑ Occur while the program is running if the environment detects an operation that is impossible to carry out.
  - ❑ Logic errors
    - ❑ Occur when a program doesn’t perform the way it was intended to.

EXCEPTIONS

- **What is an exception**
  - A situation leading to an impossibility of finishing an operation

- **How to handle an exception**
  - Provide mechanism that allows communication between the method that is detecting an exceptional condition, while is performing an operation, and the functions/objects/modules that are clients of that method and wish to handle dynamically the situation
  - Exception handling systems
    - Allows user to signal exceptions and associate handlers (set system into a coherent state) to entities

JAVA EXCEPTIONS

- **Java exception**
  - Is an object that describes an error condition occurred in the code

- **What happens when a exception occurs**
  - An object representing that exception is created and thrown in the method that caused the exception.
  - That method may choose to handle the exception itself, or pass it on.
  - Exceptions break the normal flow of control. When an exception occurs, the statement that would normally execute next is not executed.

- **At some point, the exception should be caught and processed.**

THROWING EXCETIONS

- Use the throw statement to ***throw*** an exception object

- **Example**

```
public class BankAccount {
    public void withdraw(double amout) {
            if (amount > balance)    {
                    IllegalArgumentException ex
                          = new IllegalArgumentException (
                                Amount exceeds balance");
                    throw ex;
            }
            balance = balance - amount;
    }
  }
```

THROWING EXCETIONS

- **When an exception is thrown, the current method terminates immediately.**

- **Recommendations**
  - Throw exceptions only in exceptional cases.
  - Do not abuse of exception throwing
    - Don’t use exception just to exit a deeply nested loop or a set of recursive method calls.

TREATING EXCEPTIONS

- **Every exception should be handled**

- **If an exception has no handler**
  - An error message is printed, and the program terminates.

- **A method that is ready to handle a particular exception type**
  - Contains the statements that can cause the exception inside a try block, and the handler inside a catch clause

TREATING EXECEPTIONS

- ❑ **Example**

```
try {
 System.out.println(“What is your name?”);
 String name = console.readLine();
 System.out.println(“Hello. “ + name + “!”);

} catch(IOException ex){
  // should handle exception
  ex.printStackTrace();
  System.exit(1);
}
```

Interrupts the program

Display the call stack for the method that throwed the exception

EXCEPTIONS FLOW

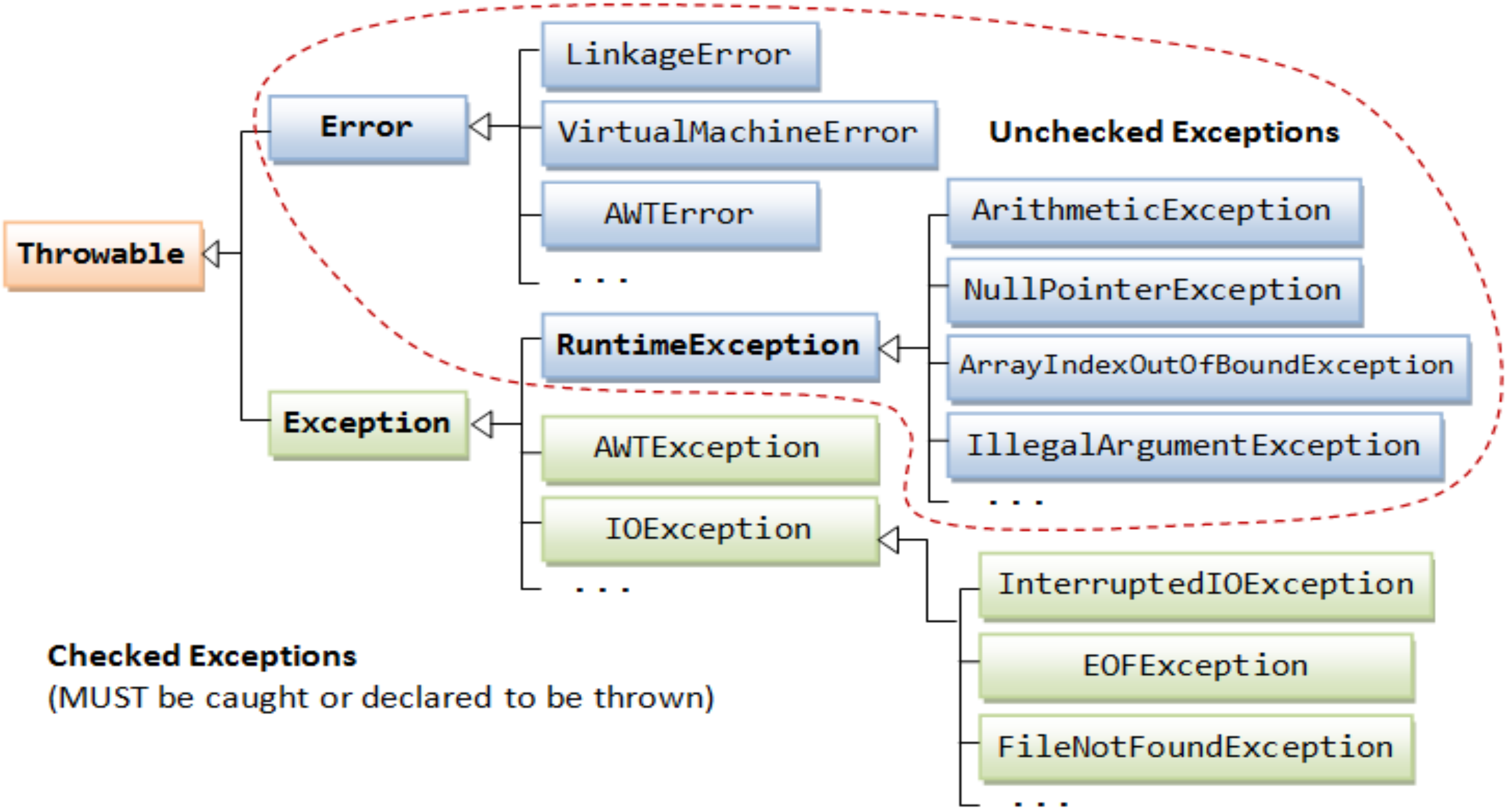
- **What happens instead depends on**
  - Whether the exception is caught
  - Where it is caught
  - What statements are executed in the ‘catch block’
  - Whether you have a ‘finally block’

EXCEPTIONS HIERACHY

- **Java organizes exceptions in inheritance tree**
  - `Throwable`
    - Superclass for all exceptions
  - `Error`
    - Are usually thrown for more serious problems, such as `OutOfMemoryError`, that may not be so easy to handle
  - `Exception`
    - `RuntimeException`
    - `TooManyListenersException`
    - `IOException`
    - `AWTException`

- **Remark**
  - The code you write should throw only exceptions, not errors.
  - Errors are usually thrown by the methods of the Java API, or by the Java virtual machine itself.

EXCEPTIONS HIERACHY



Checked Exceptions
(MUST be caught or declared to be thrown)

EXCEPTIONS HIERACHY

- ❑ **Exceptions Type**
  - ❑ Unchecked exceptions
    - ❑ `Error` and `RuntimeException`
    - ❑ Are not checked by the compiler, and hence, need not be caught or declared to be thrown in your program
  - ❑ Checked exceptions
    - ❑ They are checked by the compiler and must be caught or declared to be thrown

CATCHING AN EXCEPTION

- ❑ **Synatax**

```
try {
     // statement that could throw an exception
} catch (<exception type> e) {
     // statements that handle the exception
} catch (<exception type> e) {
     // higher in hierarchy
     // statements that handle the exception
} finally {
     // release resources
}
```

- ❑ **At most one catch block executes**

- ❑ **`finally` block always executes once, whether there’s an error or not**

CATCHING AN EXCEPTION

- ❑ **When an exception occurs, the nested `try/catch` statements are searched for a `catch` parameter matching the exception class**

- ❑ **A parameter is said to match the exception if it**
  - ❑ is the same class as the exception;
  - ❑ is a superclass of the exception;
  - ❑ if the parameter is an interface, the exception class implements the interface.

- ❑ **The first `try/catch` statement that has a parameter that matches the exception has its catch statement executed.**

- ❑ **After the catch statement executes, execution resumes with the `finally` statement, then the statements after the `try/catch` statement.**

CATCHING AN EXCEPTION

- **Catching more than one type of exception with one exception handler**
  - from Java 1.7
  - single catch block can handle more than one type of exception
  - separate each exception type with a vertical bar (|)
  - Useful
    - same behavior for multiple catch
  - Example

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

THROWING EXCEPTIONS

- Syntax
  - from method body
    - `throw new Exceprion()`
  - method prototype
    - `throws Exception1, Exception2, ..., ExceptionN`

- **If a method body throws an exception and is not threated in the body the thrown exception has to be added at method prototype**

- **Example**

```
public void foo(int i)
          throws IOException, RuntimeException {
    if ( i == 1) throw new IOException();
    if ( i == 2) throw new RuntimeException();
    System.out.println("No exeception is thrown");
}
```

TRY-WITH-RESOURCES STATEMENT

- **`try` statement that declares one or more resources**

- **A resource is an object that must be closed after the program is finished with it.**
  - Any object that implements `java.lang.AutoCloseable`, which includes all objects which implement `java.io.Closeable`

- **Syntax**

```
try (/*Resourse declaration and
initialization*/){
   //resource utilization
} catch(Exception e) { .. }
```

TRY-WITH-RESOURCES STATEMENT

- **Example**
  - before java 1.7

```
static String readFirstLineFromFileWithFinallyBlock(String
path) throws IOException {
    BufferedReader br = new BufferedReader(
                                     new FileReader(path));
    try {
        return br.readLine();
    } finally {
        if (br != null) br.close();
    }
}
```

  - java 1.7

```
static String readFirstLineFromFile(String path) throws
IOException {
    try (BufferedReader br =
                   new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

The resource is closed automatically does not have to be closed manually

CUSTOM EXCEPTION CLASS

❑ **For example if we want to withdraw mony from an accout**

```
public class BankAccount {
  public void withdraw(double amout) {
        if (amount > balance){
              IllegalArgumentException ex
              = new IllegalArgumentException (
                    Amount exceeds balance”);
              throw ex;
        } balance = balance - amount;
  }
}
```

❑ **What if we would like to throw a more specific error for the application?**

CUSTOM EXCEPTION CLASS

- **How define a custom exception class**
  - Define a class that extends `Exception`
  - Add constructors
    - default
    - one parameter: the error message
    - two parameters: the error message, an another Exception
  - Add other elements that help to explain better the exception

- **Example**

```
public class MyException extends Exception{
     public MyException(){super();}
     public MyException(String msg){super(msg);}
     public MyException(String msg, Exception e){
               super(msg,e);
     }
}
```

CUSTOM EXCEPTION CLASS

- **When to create custom exception classes?**
  - Use exception classes offered by API whenever possible
  - Write your exception class if
    - You need an exception type that is not represented by those in Java platform
    - It helps users if they could differentiate your exceptions from those thrown by classes written by other vendors
    - You want to pass more than just a string to the exception handler

INFORMATION ABOUT THROWN EXCEPTIONS

- ❑ `getMessage()`
  - ❑ Returns the detail message string of this throwable.
- ❑ `printStackTrace()`
  - ❑ Prints this throwable and its stacktrace to the standard error stream.
- ❑ `printStackTrace(PrintStream s)`
  - ❑ Prints this throwable and its stacktrace to the specified print stream.
- ❑ `printStackTrace(PrintWriter s)`
  - ❑ Prints this throwable and its stacktrace to the specified print writer.

INFORMATION ABOUT THROWN EXCEPTIONS

Example

```
public class BankDemo {
  public static void main(String [] args) {
     CheckingAccount c = new CheckingAccount(101);
     System.out.println("Depositing $500...");
     c.deposit(500.00);
     try {
        System.out.println("\nWithdrawing $100..."); c.withdraw(100.00);
        System.out.println("\nWithdrawing $600..."); c.withdraw(600.00);
     } catch (InsufficientFundsException e) {
        System.out.println("Sorry, but you are short $" + e.getAmount());
        e.printStackTrace();
     }
  }
}
```

Output

```
Depositing $500...
Withdrawing $100...
Withdrawing $600...
Sorry, but you are short $200.0
InsufficientFundsException
        at CheckingAccount.withdraw(CheckingAccount.java:25)
        at BankDemo.main(BankDemo.java:13)
```

Error stack

ASSERTIONS

- **An assertion is a Boolean expression that is placed at a point in code where is expect something to be true**

- **Syntax**
  - `assert boolean_expression;`
  - `assert boolean_expression: error_message;`

- **Behaviour**
  - If assertions are disabled, Java skips the assertion and goes on in the code.
  - If assertions are enabled and the boolean expression is true , then the assertion has been validated and nothing happens. The program continues to execute in its normal manner.
  - If assertions are enabled and the boolean expression is false, then the assertion is invalid and a `java.lang.AssertionError` is thrown.

ENABLING ASSERTIONS

- **Enabling Assertions**
  - java -enableassertions MyClass
  - java -ea MyClass

- **Example**

```
public class TestSeasons {
  public static void test(Seasons s) {
    switch (s) {
      case SPRING:
      case FALL:
        System.out.println("Shorter hours");
        break;
     case SUMMER:
        System.out.println("Longer hours");
        break;
     default:
        assert false: "Invalid season";
}}}
```

ASSERTIONS. REMARKS

- **Do not use assertions to check for valid arguments passed in to a method. Use an `IllegalArgumentException` instead**

- **Because assertions can, should, and probably will be turned off in a production environment, your assertions should not contain any business logic that affects the outcome of your code.**

  - The following assertion is not a good design because it alters the value of a variable:

```
int x = 10;
assert ++x > 10; // Not a good design!
```

NEXT COURSE PRESENTATION

- ❑ **1 Student**
  - ❑ 0.5 points bonus points at final exam
  - ❑ Presentation for next course (when the course start) regarding
    - ❑ Exceptions and lambda functions
    - ❑ Exceptions and streams
  - ❑ The presentation must be sent by email to me until Saturday for initial review
  - ❑ Express your intention now