CSD-4464 Java EE

Class 3: Exceptions, Lambdas, and Optionals

Lambton College

Exceptions, Exceptions, Exception

- An **exception** is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time, that disrupts the normal flow of the program's instructions
- Java has two types of exceptions, Checked and Unchecked exceptions

Checked Exceptions

- A checked exception is an exception that is checked (notified) by the compiler at compilation-time, these are also called compile time exceptions.

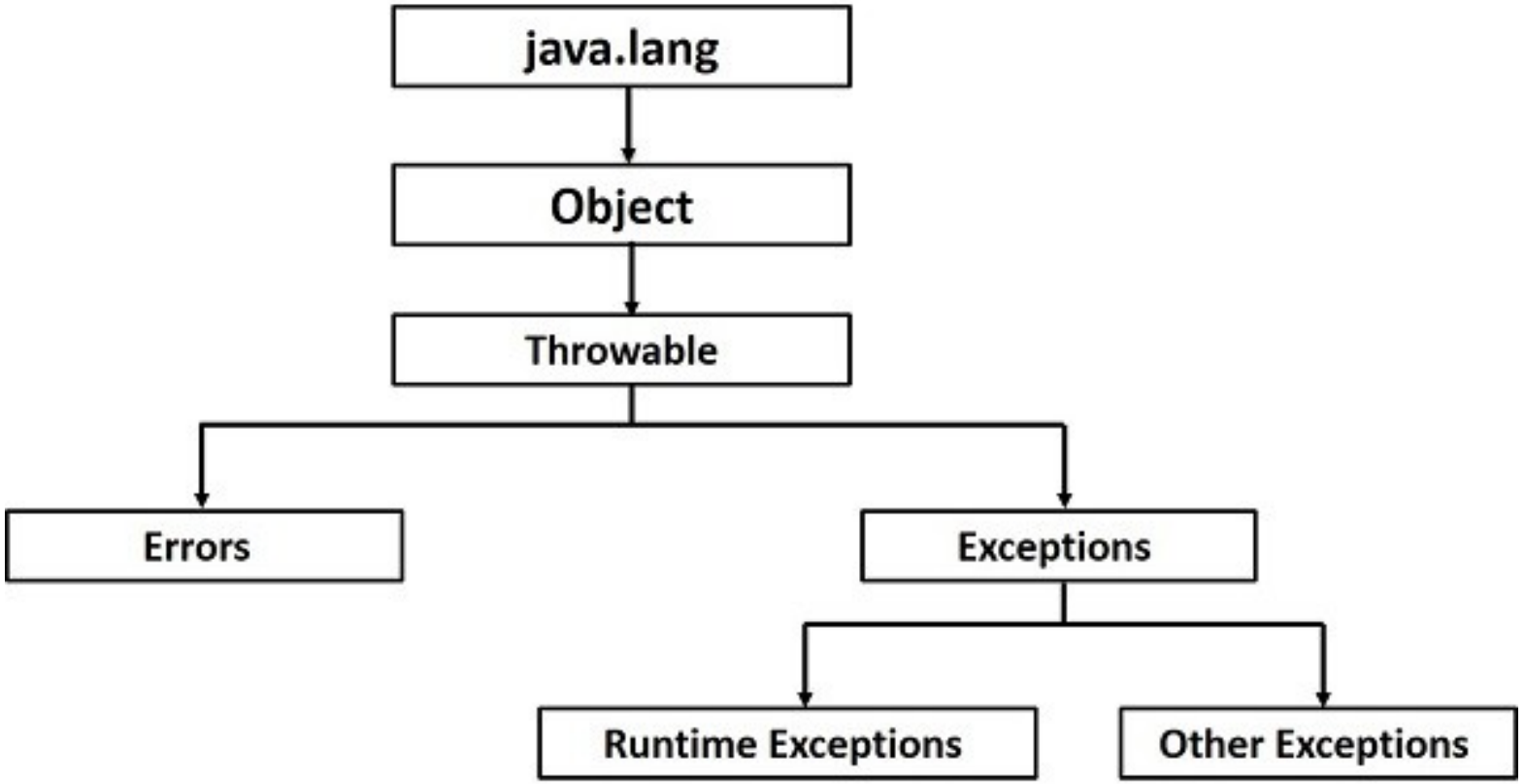
- These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions.

Unchecked Exceptions

- **Unchecked exceptions** – An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

- Examples – *ArrayIndexOutOfBoundsException*, *StackOverflowException*

Exception Hierarchy



Exceptions

- Exceptions are objects that extend throwable
- getMessage() getCause() toString() printStackTrace() getStackTrace() fillInStackTrace()
- You can throw exceptions by using the **throw** keyword and passing an object that extends throwable

Catching + Handling Exceptions

- Use Try / Catch / Finally
- The try block cannot be present without either catch clause or finally clause.
- Code cannot be present in between the try, catch, finally blocks.
- Finally is optional if Catch is present
- A catch clause cannot exist without a try statement.

Try block

- Wraps the code that you believe may throw the exception

- If the code throws an exception, java then moves to the catch block

```
Try {
    yourMethodThatMayThrowAnException()
}
```

Catch Block

- Can catch multiple types of exceptions, make sure to be as specific as possible

- Used for code that will only execute if an exception occurs

Finally Block

- The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.

- Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected

Try-with-resources

- You can initialize closeable resources inside the Try ()
- **try-with-resources**, also referred as **automatic resource management**, is a new exception handling mechanism that was introduced in Java 7, which automatically closes the resources used within the try catch block.

```
try(FileReader fr = new FileReader("file path")) {
        // use the resource
 } catch () {
      // body of catch
}
```

Lambton College

Lambdas

- Lambda expressions are introduced in Java 8 and are touted to be the biggest feature of Java 8. Lambda expression facilitates functional programming, and simplifies the development a lot.
- Using lambda expression, you can refer to any final variable or effectively final variable. Lambda expression throws a compilation error, if a variable is assigned a value the second time
- Syntax == Parameter(s) -> expression body
- Your soon to be best friend

Lambda properties

- **Optional type declaration** – No need to declare the type of a parameter. The compiler can inference the same from the value of the parameter.
- **Optional parenthesis around parameter** – No need to declare a single parameter in parenthesis. For multiple parameters, parentheses are required.
- **Optional curly braces** – No need to use curly braces in expression body if the body contains a single statement.
- **Optional return keyword** – The compiler automatically returns the value if the body has a single expression to return the value. Curly braces are required to indicate that expression returns a value.

Types of Functions

Function	//method that takes one input and returns a value
BiFunction	//method that takes two inputs and returns a value
Supplier	//method that takes no inputs and returns a value
Callable	//method that takes no inputs and returns nothing
Predicate	//method that takes one input and returns true/false

```
Integer multiply(Integer a, Integer b) {
    return a * b;
}
Integer results = multiply(2, 4);
```

Is the same as

```
BiFunction<Integer, Integer, Integer> multiply = (a, b) -> a * b
Integer results = multiply.apply(2, 4);
```

Is the same as

```
BiFunction<Integer, Integer, Integer> multiply = (a, b) -> {
    return a * b
}
Integer results = multiply.apply(2, 4);
```

Optionals – replacing null with empty!

- Optional is a container object used to contain not-null objects. Optional object is used to represent null with absent value.
- This class has various utility methods to facilitate code to handle values as ‘available’ or ‘not available’ instead of checking null values.
- Allows for a more functional approach to programming

- Syntax

Optional<Type> variableName = Optional.ofNullable(value)

Optionals methods

.get() // returns the value of the optional, throws exception if empty

.orElse(valueB**)** //returns the value of the optional, or valueB if empty

.orElseThrow(exception**)** //returns the value of the optional, or throws given exception if empty

.isPresent() //returns true or false if the optional has a value

.map() //If a value is present, apply the provided mapping function to it

.filter() //If a value is present, and the value matches the given predicate, keeps the value in the optional

https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html

```
Integer getAgeInDogYears(Integer age) {
If (age == null) {
    return 1;
}
return age * 7;
}
```

Is the same as

```
Integer getAgeInDogYears(Optional<Integer> age) {
   return age.map(a -> a * 7).orElse(1);
}
```