

## Java Core

# Exception

# Agenda

- Exception in Java
- Exception Class Hierarchy
- Exception Handling
- Statements throws and throw
- Creating own Exception
- Stack Trace
- Practical tasks



# Errors are Natural

- Any software solution faces errors: invalid user input, broken connection or bugs in code
- Errors break normal flow of the program execution and may lead to fatal results in case if not handled properly
- General Kinds of Programming Errors
  - **Compilation Errors** - prevent program from running
  - **Run-time errors** - occur while program runs
  - **Logic Errors** - prevent program from doing what it is intended to do



# Exception in Java

Lots of error checking in code makes the code harder to understand

- more complex
- more likely that the code will have errors!

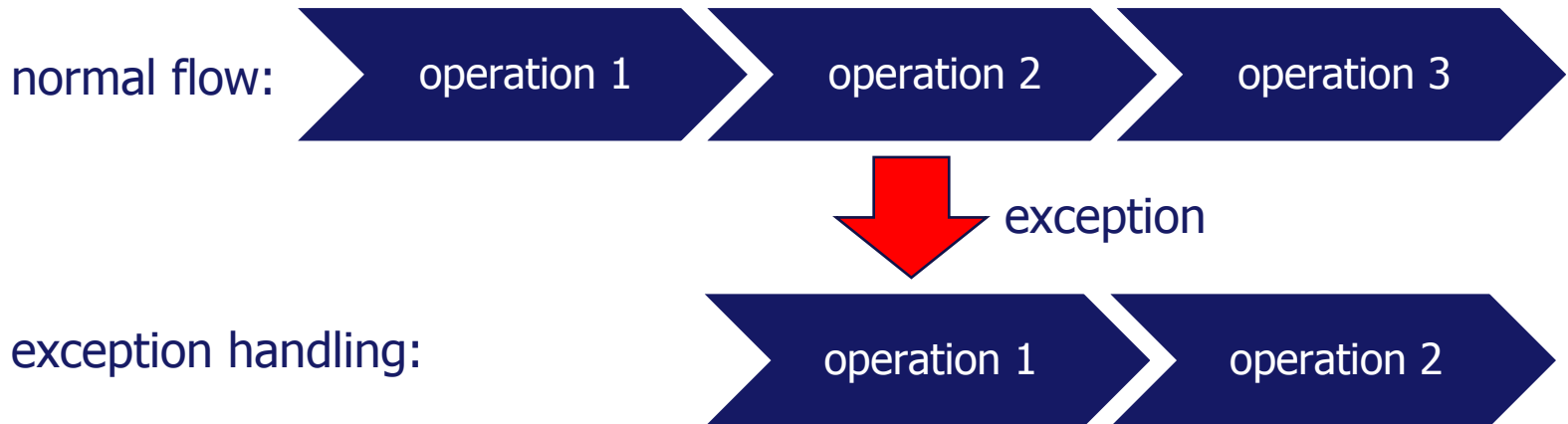
Add error checking to the following code

```
BufferedReader br = new BufferedReader(  
    new InputStreamReader(System.in));  
int k = Integer.parseInt(br.readLine( )); // ???  
  
int i = 4; int j = 0;  
System.out.println("Result: "+ (i / j)); // ???  
  
int[ ] a = new int[2];  
a[2] = 0; // ???
```

# What is Exception and Exception Handling?

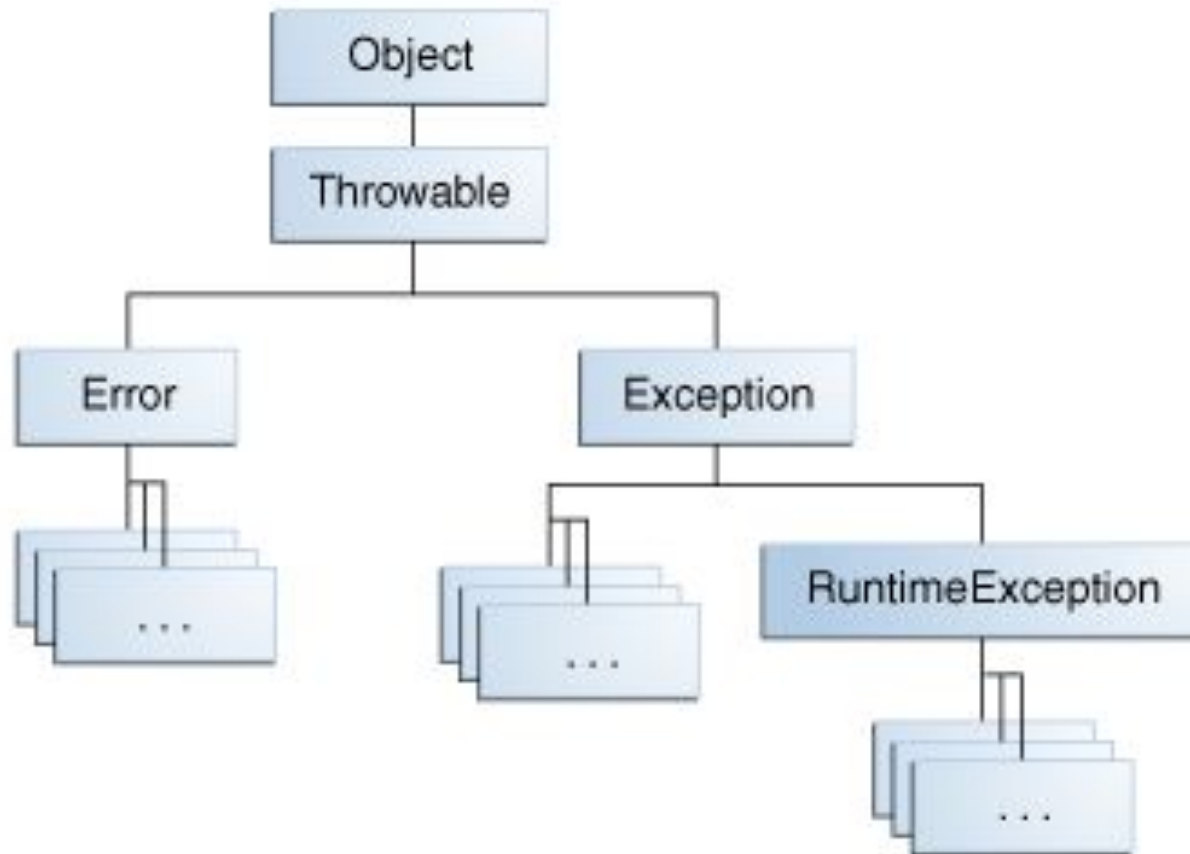
**Exception** – is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

**Exception handling** is convenient way to handle errors



# Exception Class Hierarchy

*Separate* the error checking code from the main program code - the standard approach since the 1980's



# Exception Class Hierarchy

**Exceptions** are the result of problems in the program.

**Errors** represent more serious problems associated with the **JVM-level** problems.

Exceptions are divided into three types:

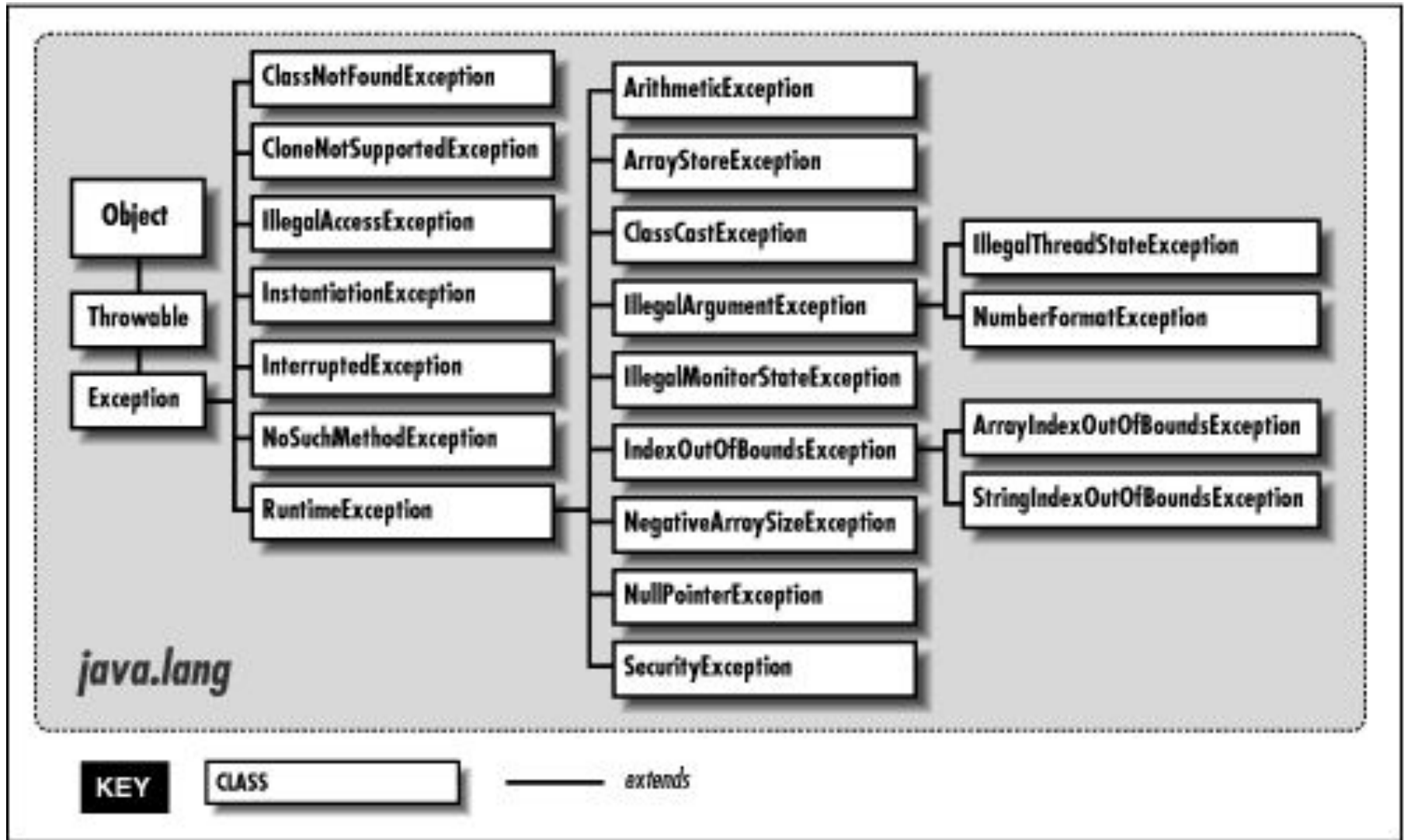
- Checked exceptions;
- Unchecked exceptions, include Errors;
- RuntimeExceptions, a subclass of Exception.

Checked exceptions are errors that *can* and *should be* **handled in the program**.

- This type includes all subclasses of Exception (but not RuntimeException).

Unchecked exceptions does **not require** mandatory handling.

# Exception Class Hierarchy (not complete)





# Exception Class Hierarchy

## 1. Checked exceptions

- subclasses of `Exception`
- recovery should be possible for these types of errors
- your code *must*
  - include *try-catch* blocks for these or the compiler will reject your program (e.g. `IOException`)
  - add *throws* to method declaration

## 2. Unchecked exceptions

- subclasses of `RuntimeException`
- exceptions of this type usually mean that your program should terminate
- the compiler does not force you to include try-catch blocks for these kinds of exceptions (e.g. `ArithmeticException`)

# Exception Handling

There are five key words in Java for working with exceptions:

- **try** - this keyword is used to mark the beginning of a block of code that can potentially lead to an error.
- **catch** - keyword to mark the beginning of a block of code designed to intercept and handle exceptions.
- **finally** - keyword to mark the beginning of a block of code, which is optional. This block is placed after the last block 'catch'. Control is usually passed to block 'finally' in any case.
- **throw** - helps to generate exceptions.
- **throws** - keyword that is prescribed in the method signature, and is indicating that the method could potentially throw an exception with the specified type.

# Exception Handling

- The programmer wraps the error-prone code inside a try block.
- If an exception occurs anywhere in the code inside the **try** block, the **catch** block is executed immediately
  - the block can use information stored in the `e` object
- After the **catch** block (the catch *handler*) has finished, execution continues after the **catch** block (in more-statements).
  - execution does **not** return to the **try** block
- If the **try** block finishes successfully without causing an exception, then execution skips to the code after the **catch** block

# Exception in Java

Java uses *exception handling*

Format of code:

```
statements;  
try {  
    code...;  
}  
catch (Exception-type e) {  
    code for dealing with e exception  
}  
more-statements;
```

a try block



a catch block

# Exception Handling

```
int doSomething(int n) {  
    try {  
        // If n = 0, then causes ArithmeticException  
        return 100 / n;  
    } catch (ArithmeticException e) {  
        // catch exception by class name  
        System.out.println("Division by zero");  
        return 0;  
    }  
}
```

# Many catch blocks

Code fragment may contain several problem places.

For example, except for **division by zero** error is possible **array indexing**.

Need to create **two** or more operators **catch** for each type of exception.

- They are checked in order.
- If an exception is detected at the first processing unit, it will be executed, and the remaining checks will be **skipped**.

If using multiple operators catch handlers **subclasses** exceptions should be **higher** than their handlers **superclasses**.

# Exception Handling

```
try {  
    // Malicious code  
} catch (ExceptionType1 e1) {  
    // Exception handling for the class ExceptionType1  
} catch (ExceptionType2 e2) {  
    // Exception handling for the class ExceptionType2  
} catch (Exception allAnotherExceptions) {  
    // Handle all exceptions previously untreated  
} catch (Throwable allAnotherErrorsAndExceptions) {  
    /* Process all errors and exceptions that have  
    not been treated so far. Bad because  
    it also handled Error- classes */  
}
```

# Exception Handling

The new design is now available in Java 7, which helps you to catch a few exceptions with one catch block :

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

This is useful when error handling is no different.



# Exception Handling

```
public int div() {  
    BufferedReader br = new BufferedReader(  
        new InputStreamReader(System.in));  
    try {  
        int n = Integer.parseInt(br.readLine());  
        int k = Integer.parseInt(br.readLine());  
        return n / k;  
    } catch (NumberFormatException | IOException e) {  
        return -1;  
    } catch (ArithmeticException e) {  
        return -2;  
    } catch (Exception e) {  
        return -3;    }  
}
```

**This code does not compile**

```
try {  
    ...  
} catch (Exception e) {  
    return -1;  
}  
catch (ArithmeticException e) {  
    return -2;  
}
```

# Finally

- A finally clause is executed even if a return statement is executed in the try or catch clauses.
- An uncaught or nested exception still exits via the finally clause.
- Typical usage is to free system resources before returning, even after throwing an exception (close files, network links)

```
try {  
    // Protect one or more statements here  
}  
catch(Exception e) {  
    // Report from the exception here  
}  
finally {  
    // Perform actions here whether  
    // or not an exception is thrown  
}
```

# Java 7 Resource Management

Java 7 has introduced a new interface `java.lang.AutoCloseable` which is extended by `java.io.Closeable` interface. To use any resource in try-with-resources, it must implement `AutoCloseable` interface else java compiler will throw compilation error.

```
public class MyResource implements AutoCloseable{
    @Override
    public void close() throws Exception {
        System.out.println("Closing");
    }
}
```

```
try (MyResource sr =
        new MyResource()) {
    //doSomething with sr
}
```

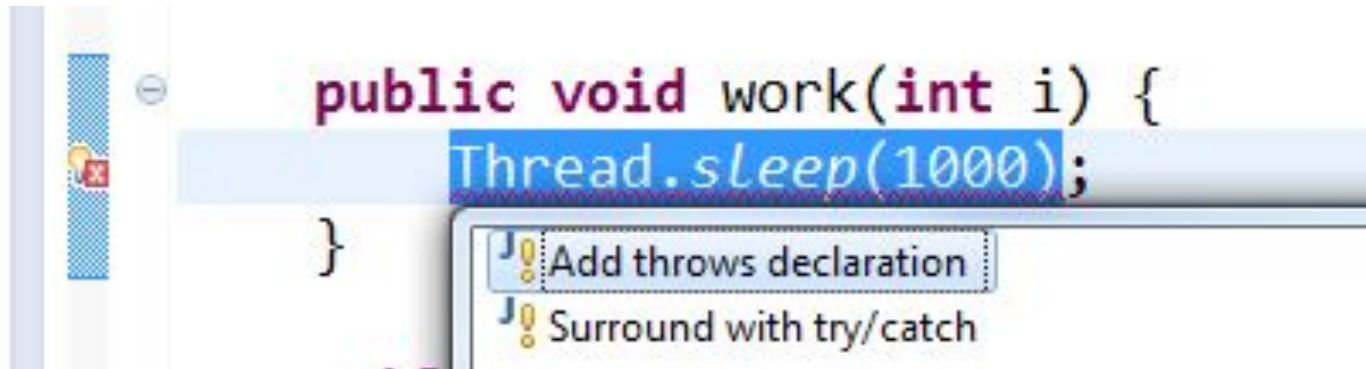
```
MyResource sr =
    new MyResource();
try {
    //doSomething with sr
} finally {
    if (sr == null) {
        sr.close();
    }
}
```

# Statement throws

If a method can throw an exception, which he does not **handle**, it must specify this behavior so that the calling code could take care of this exception.

Also there is the design **throws**, which lists the types of exceptions.

- Except Error, RuntimeException, and their subclasses.



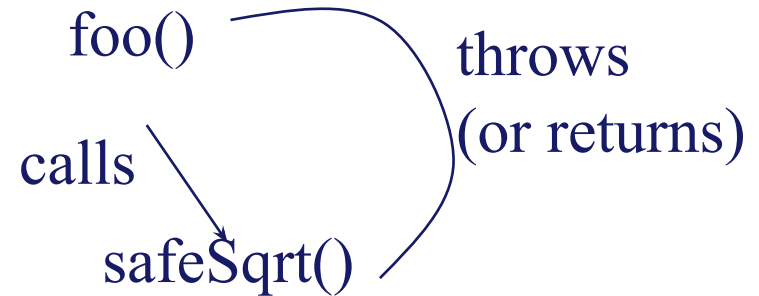
# Statement throws

For example

```
double safeSqrt(double x)
    throws ArithmeticException {
    if (x < 0.0)
        throw new ArithmeticException();
    return Math.sqrt(x);
}
```

# Statement throws

```
void foo(double x) {  
    double result;  
    try {  
        result = safeSqrt(x);  
    } catch (ArithmeticException e) {  
        System.out.println(e);  
        result = -1;  
    }  
    System.out.println("result: " + result);  
}
```




# Statement throw

You can throw exception using the **throw** statement

```
try {  
    MyClass myClass = new MyClass( );  
    if (myClass == null) {  
        throw new NullPointerException("Messages");  
    }  
} catch (NullPointerException e) {  
    // TODO  
    e.printStackTrace( );  
    System.out.println(e.getMessage( ));  
}
```

# Summary: Dealing with Checked Exceptions

```
public static void main (String[] args) {  
  
    FileReader fr = new FileReader("text.txt");  
  
}
```



```
public static void main (String[] args) {  
  
    try {  
        FileReader fr = new FileReader("text.txt");  
    } catch (FileNotFoundException e) {  
        // Do something  
    }  
  
}
```

```
public static void main (String[] args)  
    throws FileNotFoundException {  
  
    FileReader fr = new FileReader("text.txt");  
  
}
```



# Defining new exception

- You can subclass RuntimeException to create new kinds of unchecked exceptions.
- Or subclass Exception for new kinds of checked exceptions.
- Why? To improve error reporting in your program.

# Creating own checked exception

Create *checked* exception – MyException

// Creation subclass with two constructors

```
class MyException extends Exception {
```

```
    // Classic constructor with a message of error
```

```
    public MyException(String msg) {
```

```
        super(msg);
```

```
    }
```

```
    // Empty constructor
```

```
    public MyException() { }
```

```
}
```

# Creating own checked exception

```
public class ExampleException {  
    static void doSomething(int n) throws MyException {  
        if (n > 0) {  
            int a = 100 / n;  
        } else {  
            // Creation and call exception  
            throw new MyException("input value is below zero!");  
        }  
    }  
  
    public static void main(String[ ] args) {  
        try {  
            // try / catch block is required  
            doSomething(-1);  
        } catch (MyException e1) {  
            System.err.print(e1);  
        }  
    }  
}
```

# Creating own unchecked exception

If you create your own exception class from *RuntimeException*, it's not necessary to write exception specification in the procedure.

```
class MyException extends RuntimeException { }

public class ExampleException {
    static void doSomething(int n) {
        throw new MyException( );
    }
    public static void main(String[ ] args) {
        DoSomething(-1); // try / catch do not use
    }
}
```

# Limitation on overridden methods

- Overridden method can't change list of exceptions declared in throws section of parent method
- We can add new exception to child class when it is a descendant of an exception from the parent class or it is a runtime exception

```
public class Base {  
    public void doSomething() throws IllegalAccessException {}  
}
```

```
public class Child extends Base {  
    @Override  
    public void doSomething() throws NoSuchMethodException {}  
}
```



# Stack Trace

The exception keeps being passed out to the next enclosing block until:

- a suitable handler is found;
- or there are no blocks left to try and the program terminates with *a stack trace*

If no handler is called, then the system prints a *stack trace* as the program terminates

- it is a list of the called methods that are waiting to return when the exception occurred
- *very useful* for debugging/testing

The stack trace can be printed by calling `printStackTrace()`

# Stack Trace

```
public static void method1() throws MyException {  
    method2();  
}  
public static void method2() throws MyException {  
    method3();  
}  
public static void method3() throws MyException {  
    throw  
    new MyException("Exception thrown in method3" );  
}  
} // end of UsingStackTrace class
```

# Stack Trace

- `method1 ()` and `method2 ()` require **throws** declarations since they call a method that *may* throw a `MyException`.
- The compiler will reject the program at compile time if the **throws** are not included
  - Exception is a non-runtime (checked) exception



# Using a Stack Trace

```
// The getMessage and printStackTrace methods
public static void main( String[] args) {
    try {
        method1();
    } catch (Exception e) {
        System.err.println(e.getMessage() + "\n");
        e.printStackTrace();
    }
}
```

# Using a Stack Trace

main()

method1(  
)

method2(  
)

method3(  
Exception!!  
)

e.getMessage() output

```
> java UsingStackTrace
Exception thrown in method3

java.lang.Exception: Exception thrown in method3
    at UsingStackTrace.method3(UsingStackTrace.java:29)
    at UsingStackTrace.method2(UsingStackTrace.java:26)
    at UsingStackTrace.method1(UsingStackTrace.java:23)
    at UsingStackTrace.main(UsingStackTrace.java:13)
>
```

e.printStackTrace()  
output

# Exception Handling Best Practices

- **Use Specific Exceptions** – we should always *throw* and *catch* specific exception classes so that caller will know the root cause of exception easily and process them. This makes debugging easy and helps client application to handle exceptions appropriately.
- **Throw early** - we should try to throw exception as early as possible.
- **Catch late** – we should catch exception only when we can handle it appropriate.
- **Close resources** - we should close all the resources in finally block or use Java 7 block try-with-resources.
- Do Not Use Exceptions to Control Application Flow

<http://www.journaldev.com/1696/exception-handling-in-java#java-7-arm>

# Practical tasks

1. Create a method for calculating the area of a rectangle `int squareRectangle (int a, int b)`, which should throw an exception if the user enters negative value. Input values a and b from console. Check the `squareRectangle` method in the method `main`. Check to input nonnumeric value.
2. Create a class `Plants`, which includes fields `int size`, `Color color` and `Type type`, and constructor where these fields are initialized. `Color` and `type` are Enum. Override the method `toString()`. Create classes `ColorException` and `TypeException` and describe there all possible colors and types of plants. In the method `main` create an array of five plants. Check to work your exceptions.

# HomeWork (online course)

- UDEMY course "Java Tutorial for Complete Beginners":  
<https://www.udemy.com/java-tutorial/>
- Complete lessons 38-42:

## 38. Handling Exceptions

[Learn Java Tutorial for Beginners \(Video\), Part 34: Handling Exceptions](#)

## 39. Multiple Exceptions

[Learn Java Tutorial for Beginners \(Video\), Part 36: Multiple Exceptions](#)

## 40. Runtime vs. Checked Exceptions

[Learn Java Tutorial for Beginners \(Video\), Part 37: Runtime Exceptions](#)

## 41. Abstract Classes

[Learn Java Tutorial for Beginners \(Video\), Part 38: Abstract Classes](#)

## 42. Reading Files With File Reader

---

# Homework

- Create method *div()*, which calculates the dividing of two double numbers. In *main* method input 2 double numbers and call this method. Catch all exceptions.
- Write a method *readNumber(int start, int end)*, that read from console integer number and return it, if it is in the range [start...end].

If an invalid number or non-number text is read, the method should throw an exception.

Using this method write a method *main()*, that has to enter 10 numbers:

$a_1, a_2, \dots, a_{10}$ , such that  $1 < a_1 < \dots < a_{10} < 100$

- Refactor your previous homework (1-7) and try to handle all possible exceptions in your code.

# The end

**USA HQ**

Toll Free: 866-687-3588

Tel: +1-512-516-8880

**Ukraine HQ**

Tel: +380-32-240-9090

**Bulgaria**

Tel: +359-2-902-3760