JAVA klaidų apdorojimo mechanizmas

Exception Handling Keywords

- Java provides specific keywords for exception handling purposes, we will look after them first and then we will write a simple program showing how to use them for exception handling.
- **throw** – We know that if any exception occurs, an exception object is getting created and then Java runtime starts processing to handle them. Sometime we might want to generate exception explicitly in our code, for example in a user authentication program we should throw exception to client if the password is null. **throw** keyword is used to throw exception to the runtime to handle it.
- **throws** – When we are throwing any exception in a method and not handling it, then we need to use**throws** keyword in method signature to let caller program know the exceptions that might be thrown by the method. The caller method might handle these exceptions or propagate it to it's caller method using throws keyword. We can provide multiple exceptions in the throws clause and it can be used with main() method also.
- **try-catch** – We use try-catch block for exception handling in our code. try is the start of the block and catch is at the end of try block to handle the exceptions. We can have multiple catch blocks with a try and try-catch block can be nested also. catch block requires a parameter that should be of type Exception.
- **finally** – finally block is optional and can be used only with try-catch block. Since exception halts the process of execution, we might have some resources open that will not get closed, so we can use finally block. finally block gets executed always, whether exception occurred or not.

Klaidų apdorojimo pavyzdys

```java
import java.io.FileNotFoundException;
import java.io.IOException;

public class ExceptionHandling {

    public static void main(String[] args) throws FileNotFoundException, IOException {
        try{
            testException(-5);
            testException(-10);
        }catch(FileNotFoundException e){
            e.printStackTrace();
        }catch(IOException e){
            e.printStackTrace();
        }finally{
            System.out.println("Releasing resources");
        }
        testException(15);
    }

    public static void testException(int i) throws FileNotFoundException, IOException{
        if(i < 0){
            FileNotFoundException myException = new FileNotFoundException("Negative Integer "+i);
            throw myException;
        }else if(i > 10){
            throw new IOException("Only supported for index 0 to 10");
        }

    }

}
```

Programos išvestis (angl. output):

```
java.io.FileNotFoundException: Negative Integer -5
    at com.journaldev.exceptions.ExceptionHandling.testException(ExceptionHandling.java:24)
    at com.journaldev.exceptions.ExceptionHandling.main(ExceptionHandling.java:10)
Releasing resources
Exception in thread "main" java.io.IOException: Only supported for index 0 to 10
    at com.journaldev.exceptions.ExceptionHandling.testException(ExceptionHandling.java:27)
    at com.journaldev.exceptions.ExceptionHandling.main(ExceptionHandling.java:19)
```

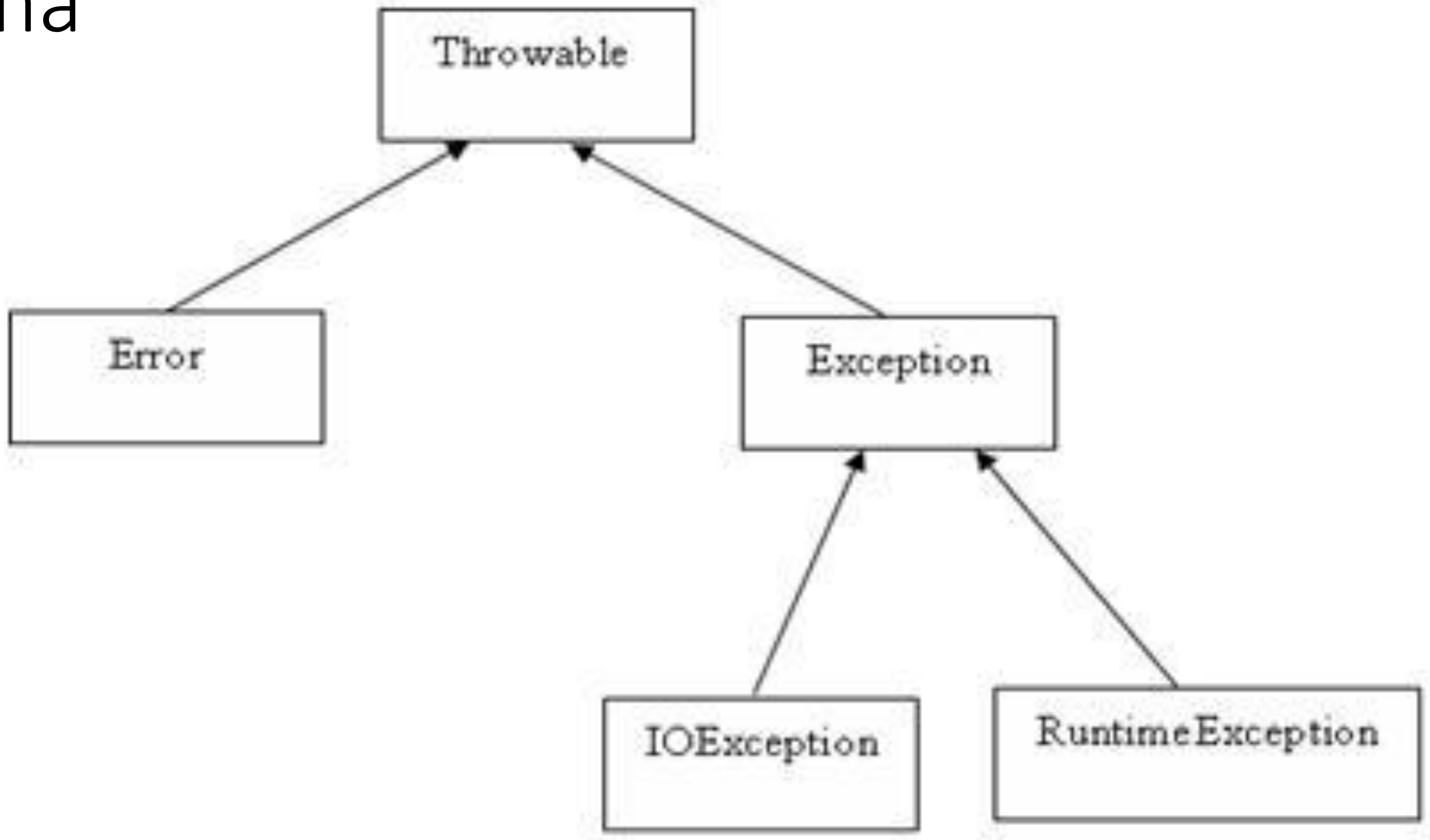
Notice that testException() method is throwing exception using throw keyword and method signature uses throws keyword to let caller know the type of exceptions it might throw. In main() method, I am handling exception using try-catch block in main() method and when I am not handling it, I am propagating it to runtime with throws clause in main method. Notice that testException(-10) never gets executed because of exception and then execution of finally block after try-catch block is executed. The printStackTrace() is one of the useful method in Exception class and used for debugging purpose.

- We can't have catch or finally clause without a try statement.
- A try statement should have either catch block or finally block, it can have both blocks.
- We can't write any code between try-catch-finally block.
- We can have multiple catch blocks with a single try statement.
- try-catch blocks can be nested similar to if-else statements.
- We can have only one finally block with a try-catch statement.

Exception Hierarchy

- As stated earlier, when any exception is raised an **exception object** is getting created. Java Exceptions are hierarchical and inheritance is used to categorize different types of exceptions. Throwable is the parent class of Java Exceptions Hierarchy and it has two child objects – Error and Exception. Exceptions are further divided into checked exceptions and runtime exception.

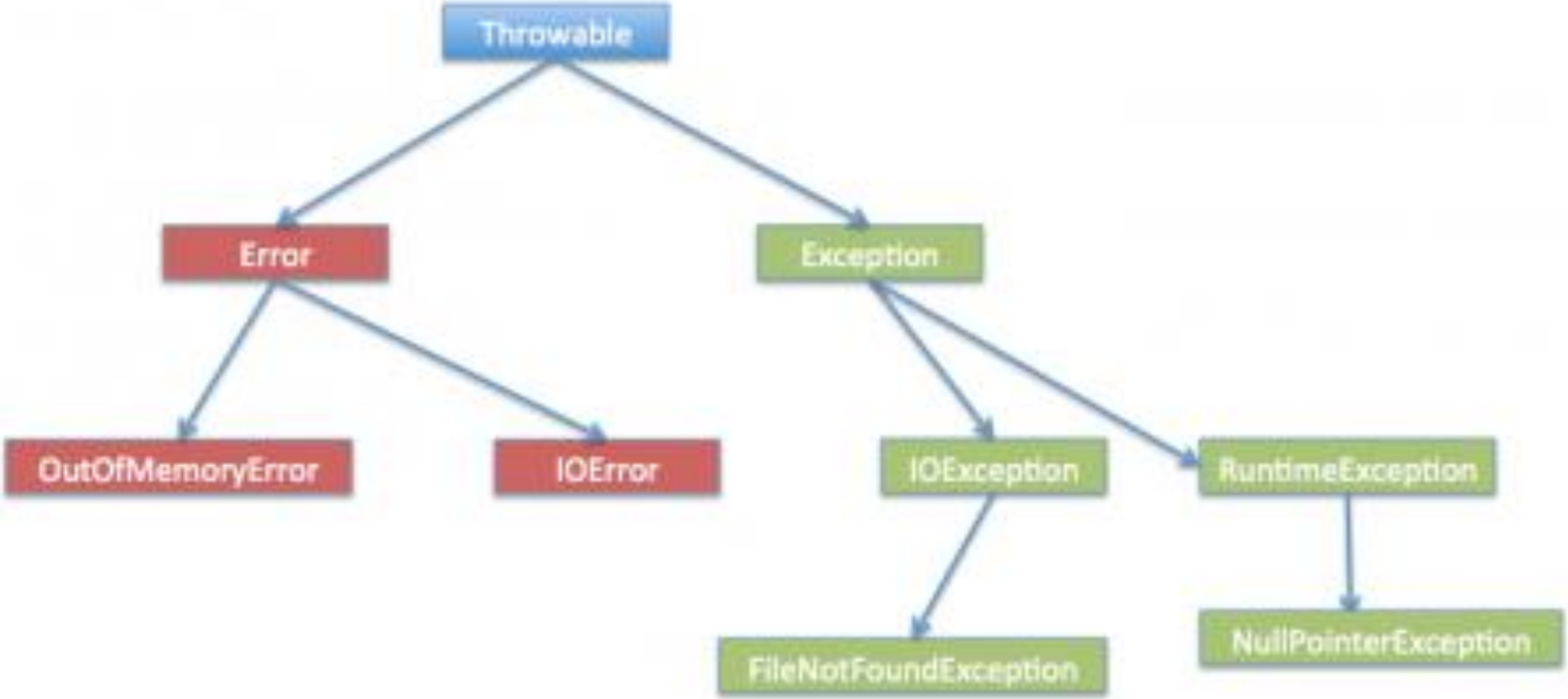
Pagrindinė klaidų apdorojimo mechanizmo schema



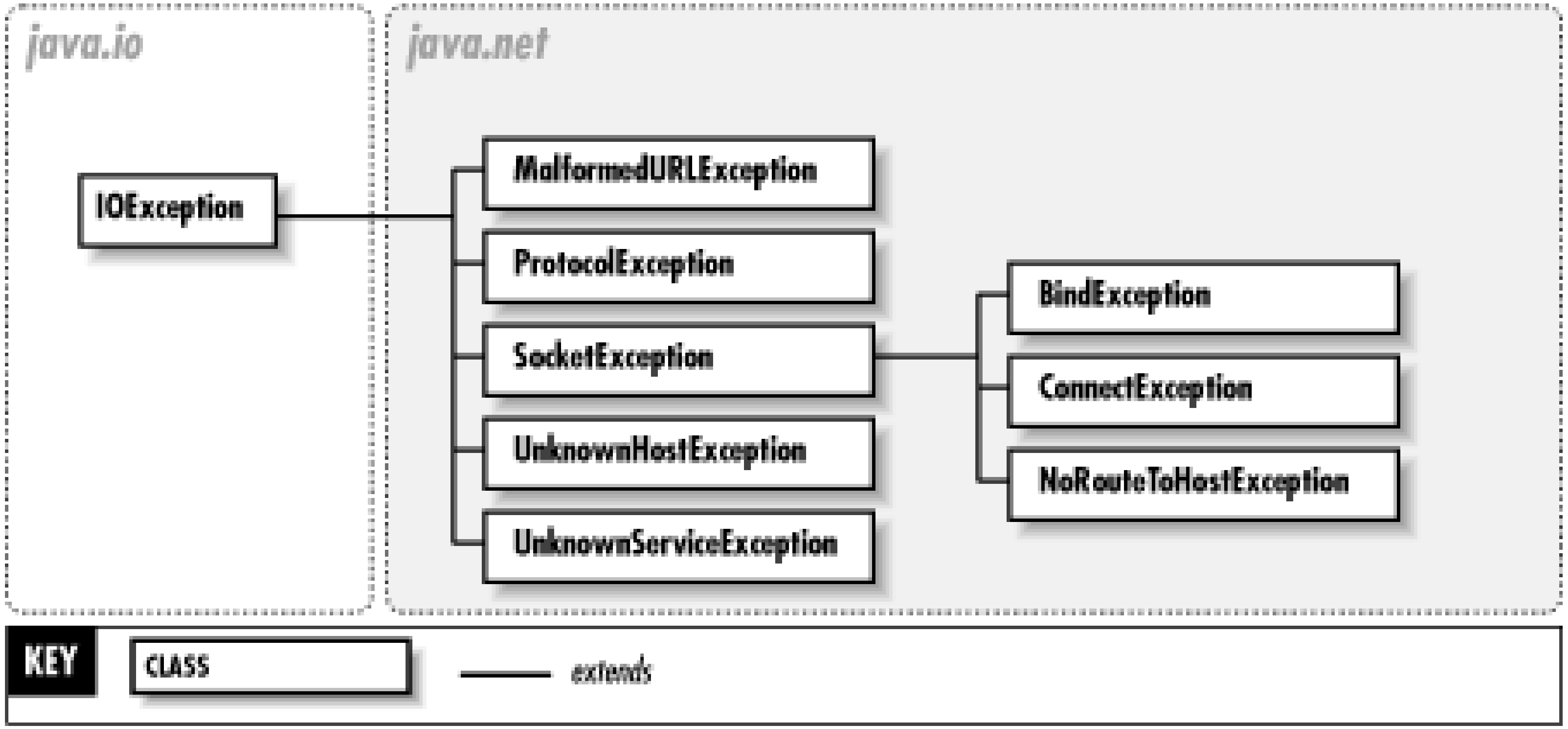
1.Errors: Errors are exceptional scenarios that are out of scope of application and it’s not possible to anticipate and recover from them, for example hardware failure, JVM crash or out of memory error. That’s why we have a separate hierarchy of errors and we should not try to handle these situations. Some of the common Errors are OutOfMemoryError and StackOverflowError.

2.Checked Exceptions: Checked Exceptions are exceptional scenarios that we can anticipate in a program and try to recover from it, for example FileNotFoundException. We should catch this exception and provide useful message to user and log it properly for debugging purpose. Exception is the parent class of all Checked Exceptions and if we are throwing a checked exception, we must catch it in the same method or we have to propagate it to the caller using throws keyword.

3.Runtime Exception: Runtime Exceptions are cause by bad programming, for example trying to retrieve an element from the Array. We should check the length of array first before trying to retrieve the element otherwise it might throw ArrayIndexOutOfBoundException at runtime. RuntimeException is the parent class of all runtime exceptions. If we are throwing any runtime exception in a method, it’s not required to specify them in the method signature throws clause. Runtime exceptions can be avoided with better programming.


Throwable
Error
Exception
OutOfMemoryError
IOError
IOException
RuntimeException
FileNotFoundException
NullPointerException

JAVA.NET Exception



Klaidų apdorojimas

- Klaida, arba išimtis (Java jas vadina *exception)* - objektas, aprašantis klaidingą situaciją programoje. Šis objektas sukuriamas kilus situacijai ir perduodamas situaciją sukėlusiam metodui. Metodas, reaguodamas į šią išimtį, gali atlikti kokius nors veiksmus - apdoroti išimtį arba perduoti išimtį apdoroti aukštesniam metodui. Išimtį programoje gali sukelti ir pats programuotojas, informuodamas apie kokią nors neleistiną situaciją (pavyzdžiui, jei koks programos kintamasis išeina už algoritmo jam leidžiamų ribų).

- Išimtims apdoroti reikalingi raktažodžiai ***try, catch, throw, throws* ir *finally.*** Raktažodis *try* figūriniais skliaustais apima operatorių bloką, kuris bus kontroliuojamas. Jei šiame bloke kyla išimtis - ją reikia „sugauti", arba perimti - ***catch,*** ir apdoroti. Programos vykdymo laiko klaidas *(**run-time errors**)* sukelia automatiškai Java.

- Programuotojas pats savo išimtis sukelia specialiu operatoriumi ***throw.*** Metodas, kuriame gali kilti išimtis, bet kuris pats neapdoroja šios išimties, o nori perduoti ją apdoroti aukštesniam metodui, turi būti pažymėtas operatoriumi ***throws.*** Operatorius ***finally*** atlieka panašias funkcijas kaip ***default*** operatoriuje ***switch**:* nurodo kodo fragmentą, kuris būtinai turi buti atliktas neatsižvelgiant į tai, kilo išimtis ar ne. Taigi principinė klaidų apdorojimo schema yra tokia:

try *{* kontroliuojamas kodo blokas *}*

catch(ExceptionType1 et1) { et1 *apdorojimas* }

catch(ExceptionType2 et2) { et2 *apdorojimas* }

finally{ butinai atliekamas kodo fragmentas *}*

Visos išimtys yra klasės ***Throwable*** subklasės. Žemiau yra dvi jos subklasės - ***Exception*** ir ***Error***. ***Exception*** klasė apima klaidingas programos situacijas, kurias turėtų sugauti ir apdoroti programuotojas. Jo kuriamos išimtys taip pat turi buti šios klasės subklasės. Svarbi ***Exception*** subklasė yra ***RuntimeException****:* ji apima tokias klaidas, kaip dalyba iš nulio, masyvo indekso išeitis už masyvo ribų ir pan. Aišku, programuotojas norėdamas gali sukelti tokio tipo išimtis ir pats. Klasės ***Error*** ir jos subklasių apibrėžiamų išimčių programoje perimti nereikia: tai tokios išimtys, kurių programa negali pati apdoroti - tai sisteminės ir kompiliavimo klaidos.

Jei ***RuntimeException*** išimties programuotojas neperima, ją automatiškai perima ***Java*** klaidų apdorojimo mechanizmas: išvedamas pranešimas apie klaidą, metodų kvietimų seka, ir programos darbas nutraukiamas. Seka išvardija visą metodų ir jų klasių vardų seką nuo išimtį sukėlusio metodo iki aukščiausio metodo ir atrodo taip:

at KlasėsVardas, MetodoVardas (RinkmenosVardas./ava: eilutės numeris)
at KlasėsVardas,AukštesniojoMetodo Vardas (RinkmenosVardas.java: eilutės
numeris)

Operatoriai ***try*** ir ***catch***

Operatoriai reikalingi, kai programuotojas pats bando perimti *Java* ar savo paties sukurtą išimtį. Pavyzdžiui, dalybos iš nulio perėmimas:

```
class MyException{
        public static void main( String args [ ] ){
        int numerator, denominator, result;
                try{
                numerator = 123; denominator = 0;
                result = numerator/denominator;
                System. out.println( "This won, 't be printed " );
                } catch(ArithmeticException ae ){
                System. out.println( "Processing of exception " );
                }
                System. out.println( "After catch of exception " );
```

Programos išvestis:

- Processing of exception
- After catch of exception

- Kai programa po kontroliuojamo bloko bando perimti kelias skirtingas išimtis - reikalingi keli skirtingi operatoriai *catch.* Jie turi būti išdėstomi tam tikra tvarka: pirmiau reikia perimti išimtį-subklasę, o vėliau - išimtį-superklasę. Antraip dalis programos kodo niekada nebūtų vykdoma, o tai jau sintaksės klaida. Pavyzdžiui:

```
try{
} catch (Exception e ){
        System.out.println( "Exception " + e);
} catch (ArithmeticException ae ){
        System.out.println( "Exception " + ae);
}
```

- ***ArithmeticException*** yra ***Exception*** subklasė, todėl visada bus perimta pirmoji išimtis - *Exception,* o antrasis blokas niekada nebūtų vykdomas. Todėl toks programos fragmentas negalėtų būti sukompiliuotas.
- Minėtieji operatoriai gali būti ir kartotiniai: vieno ***try*** bloko viduje gali būti kitas ***try*** blokas ir t. t.

Operatorius *throw*

- Sukelia programuotojo išimtį naudojant tokią sintaksę:
- . . . **throw ThrowableInstance**;
- čia ***ThrowableInstance*** yra arba vidinė *Java* išimtis - viena kuri nors ***Throwable*** klasės subklasė, arba paties programuotojo sukurta išimtis (taip pat turi paveldėti ***Throwable*** savybes). Šis objektas sukuriamas įprastai, operatoriumi **new**. Visų vidinių *Java* išimčių konstruktoriai yra dviejų tipų: be argumento arba su *String* tipo argumentu. Pastaruoju galima objektui suteikti paaiškinamąjį tekstą, kurį vėliau spausdintume metodu ***toString***.
- Įvykdžius operatorių, programos vykdymas pertraukiamas ir valdymas atiduodamas atitinkamam ***catch*** blokui. Jei toks nerandamas - programą stabdo *Java* klaidų apdorojimo mechanizmas. Pavyzdys-schema: sukeliama vidinė *Java* klaida ***NullPointerException*** - tam operatoriumi ***new*** sukuriamas jos objektas; ji sugaunama, paskui sukeliama nauja tokio pat tipo klaida, o jos apdorojimas atiduodamas aukštesniam metodui:

```
public class MyException2{

    static void throwException (){
    try{
        throw new NullPointerException ( "Example " );
    } catch(NullPointerException npe ){
        System. out.println( "Caugth inside throwException ");
        System. out.println( "Exception type: " + npe );
        throw npe;
    }
    }
    public static void main( String args [ ] ){ try{
        throwException ();
    } catch(NullPointerException npe ){
        System. out.println( "Caugth repeatedly inside main ");
    }
    }

}
```

Programos išvestis:

```
Caugth inside throwException
Exception type: Example
Caugth repeatedly inside main
```

Operatorius *throws*

- Tai raktažodis metodui, kuris gali sukelti išimtį, tačiau pats jos neapdoros - tada klaidą privalo apdoroti aukštesnis kviečiantysis metodas (aišku, šis gali peradresuoti apdorojimą dar aukštesniam metodui ir t. t.). Po operatoriaus ***throws*** galima išvardyti ir kelias išimtis - tada visos jos turi būti perimtos aukštesniame metode. Nereikia apdoroti tik ***Error*** ir ***RuntimeException*** išimčių bei jų *sub išimčių.*

```
public class MyException3{
        static void throwException() throws Exception{
        System.out.println( "Inside throwException ");
        throw new Exception ( "Example " );
        }
        public static void main( String args [ ] ) {
        try{
                throwException ();
        }
        catch(Exception e ){
                System. out.println( "Exception " + e + " caught");
        }
}
}
```

Operatorius *finally*

- Šio operatoriaus blokas rašomas po visų ***catch*** blokų ir vykdomas bet kokiu atveju - kilo išimtis ar ne. Pavyzdžiu galėtų būti duomenų skaitymas iš rinkmenos: skaitant duomenis gali kilti vidinė *Java* išimtis ***IOException*** - tokiu atveju ją reikia perimti, o po jos apdorojimo - uždaryti rinkmeną; jei duomenys nuskaityti tvarkingai be klaidų - vis tiek po visko rinkmeną reikia uždaryti. Beje, jei yra operatorius ***finally,*** *Java* kompiliatorius nereikalauja po ***try*** bloko ***catch*** operatorių. Tas pat pavyzdys su šiuo operatoriumi:

Finally operatorius kaip ir **switch** sakinio default: (reikšmė pagal nutylėjimą).

```
public class MyException4{
    static void throwException() throws Exception{
    System.out.println( "Inside throwException ");
    throw new Exception ( "Example " );
}
    public static void main( String args [ ] ) {
    try{
        throwException ();
    } finally{
        System.out.println( "An exception caught");
    }
}
}
```

Exception and all of it’s subclasses doesn’t provide any specific methods and all of the methods are defined in the base class Throwable. The exception classes are created to specify different kind of exception scenarios so that we can easily identify the root cause and handle the exception according to it’s type. Throwable class implements Serializable interface for interoperability.

Some of the useful methods of Throwable class are;

1.public String getMessage() – This method returns the message String of Throwable and the message can be provided while creating the exception through it’s constructor.

2.public String getLocalizedMessage() – This method is provided so that subclasses can override it to provide locale specific message to the calling program. Throwable class implementation of this method simply use getMessage() method to return the exception message.

3.public synchronized Throwable getCause() – This method returns the cause of the exception or null id the cause is unknown.

4.public String toString() – This method returns the information about Throwable in String format, the returned String contains the name of Throwable class and localized message.

5.public void printStackTrace() – This method prints the stack trace information to the standard error stream, this method is overloaded and we can pass PrintStream or PrintWriter as argument to write the stack trace information to the file or stream.

Klaidų apdorojimo klases galima apibrėžti ir patiems MyException.java

```
public class MyException extends Exception {

    private static final long serialVersionUID = 4664456874499611218L;

    private String errorCode="Unknown_Exception";

    public MyException(String message, String errorCode){
        super(message);
        this.errorCode=errorCode;
    }

    public String getErrorCode(){
        return this.errorCode;
    }


}
```

Apibrėžto MyException.java panaudojimas. CustomExceptionExample.java

```java
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;

public class CustomExceptionExample {

    public static void main(String[] args) throws MyException {
        try {
            processFile("file.txt");
        } catch (MyException e) {
            processErrorCodes(e);
        }

    }

    private static void processErrorCodes(MyException e) throws MyException {
        switch(e.getErrorCode()){
        case "BAD_FILE_TYPE":
            System.out.println("Bad File Type, notify user");
            throw e;
        case "FILE_NOT_FOUND_EXCEPTION":
            System.out.println("File Not Found, notify user");
            throw e;
        case "FILE_CLOSE_EXCEPTION":
            System.out.println("File Close failed, just log it.");
            break;
        default:
            System.out.println("Unknown exception occured, lets log it for further debugging."+e.getMessage());
            e.printStackTrace();
        }
    }
```

```java
 private static void processFile(String file) throws MyException {
        InputStream fis = null;
        try {
            fis = new FileInputStream(file);
        } catch (FileNotFoundException e) {
            throw new MyException(e.getMessage(),"FILE_NOT_FOUND_EXCEPTION");
        }finally{
            try {
                if(fis !=null)fis.close();
            } catch (IOException e) {
                throw new MyException(e.getMessage(),"FILE_CLOSE_EXCEPTION");
            }
        }
    }

}
```

Exception Handling Best Practices

1.Use Specific Exceptions – Base classes of Exception hierarchy doesn’t provide any useful information, thats why Java has so many exception classes, such as IOException with further sub-classes as FileNotFoundException, EOFException etc. 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.

2.Throw Early or Fail-Fast – We should try to throw exceptions as early as possible. Consider above processFile() method, if we pass null argument to this method we will get following exception:

```
Exception in thread "main" java.lang.NullPointerException
    at java.io.FileInputStream.<init>(FileInputStream.java:134)
    at javFileInputStream.java:97)
    at com.journaldev.exceptions.CustomExceptionExample.processFile(CustomExceptionExample.java:42a.io.FileInputStream.<i
    at com.journaldev.exceptions.CustomExceptionExample.main(CustomExceptionExample.java:12)
```

While debugging we will have to look out at the stack trace carefully to identify the actual location of exception. If we change our implementation logic to check for these exceptions early as below;

```
private static void processFile(String file) throws MyException {
        if(file == null) throw new MyException("File name can't be null", "NULL_FILE_NAME");
//further processing
}
```

Then the exception stack trace will be like below that clearly shows where the exception has occurred with clear message:

```
com.journaldev.exceptions.MyException: File name can't be null
    at com.journaldev.exceptions.CustomExceptionExample.processFile(CustomExceptionExample.java:37)
    at com.journaldev.exceptions.CustomExceptionExample.main(CustomExceptionExample.java:12)
```

3. Catch Late – Since java enforces to either handle the checked exception or to declare it in method signature, sometimes developers tend to catch the exception and log the error. But this practice is harmful because the caller program doesn’t get any notification for the exception. We should catch exception only when we can handle it appropriately. For example, in above method I am throwing exception back to the caller method to handle it. The same method could be used by other applications that might want to process exception in a different manner. While implementing any feature, we should always throw exceptions back to the caller and let them decide how to handle it.

4. Closing Resources – Since exceptions halt the processing of program, we should close all the resources in finally block or use Java 7 try-with-resources enhancement to let java runtime close it for you.

5. Logging Exceptions – We should always log exception messages and while throwing exception provide clear message so that caller will know easily why the exception occurred. We should always avoid empty catch block that just consumes the exception and doesn't provide any meaningful details of exception for debugging.

6. Single catch block for multiple exceptions – Most of the times we log exception details and provide message to the user, in this case we should use java 7 feature for handling multiple exceptions in a single catch block. This approach will reduce our code size and it will look cleaner too.

7. Using Custom Exceptions – It's always better to define exception handling strategy at the design time and rather than throwing and catching multiple exceptions, we can create a custom exception with error code and caller program can handle these error codes. Its also a good idea to create a utility method to process different error codes and use it.

8. Naming Conventions and Packaging – When you create your custom exception, make sure it ends with Exception so that it will be clear from name itself that it's an exception. Also make sure to package them like it's done in JDK, for example IOException is the base exception for all IO operations.

9. Use Exceptions Judiciously – Exceptions are costly and sometimes it’s not required to throw exception at all and we can return a boolean variable to the caller program to indicate whether an operation was successful or not. This is helpful where the operation is optional and you don’t want your program to get stuck because it fails. For example, while updating the stock quotes in database from a third party webservice, we may want to avoid throwing exception if the connection fails.

10. Document the Exceptions Thrown – Use javadoc @throws to clearly specify the exceptions thrown by the method, it’s very helpful when you are providing an interface to other applications to use.

Catch blokas nuo java 7 versijos atsinaujino

Java žemesnės versijos nei **7**. catch bloko apibrėžimas atrodė taip:

```
catch (IOException ex) {
     logger.error(ex);
     throw new MyException(ex.getMessage());
catch (SQLException ex) {
     logger.error(ex);
     throw new MyException(ex.getMessage());
}catch (Exception ex) {
     logger.error(ex);
     throw new MyException(ex.getMessage());
}
```

Java nuo versijos **7+**. catch bloko apibrėžimas analogiško kodo pateikto kairėje atrodo taip:

```
catch(IOException | SQLException |
Exception ex){
     logger.error(ex);
     throw new
MyException(ex.getMessage());
}
```