BBM 102 – Introduction to Programming II

Spring 2019

Exceptions

Today

- What is an exception?
- What is exception handling?
- Keywords of exception handling
  - try
  - catch
  - finally
- Throwing exceptions
  - throw
  - Custom exception classes
- Getting data from an exception object
- Checked and unchecked exceptions
  - throws

Errors

- Syntax errors
  - arise because the rules of the language have not been followed.
  - detected by the compiler.

- *Logic errors*
  - lead to wrong results and detected during testing.
  - arise because the logic coded by the programmer was not correct.

- *Runtime errors*
  - occur when the program is running and the environment detects an operation that is impossible to carry out.

Errors

- Code errors
  - Divide by zero
  - Array out of bounds
  - Integer overflow
  - Accessing a null pointer (reference)

- Programs *crash* when an exception goes <u>untrapped</u>, i.e., not handled by the program.

Runtime Errors

```
import java.util.Scanner;

public class ExceptionDemo {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter an integer: ");
    int number = scanner.nextInt();

    // Display the result
    System.out.println(
      "The number entered is " + number);
  }
}
```

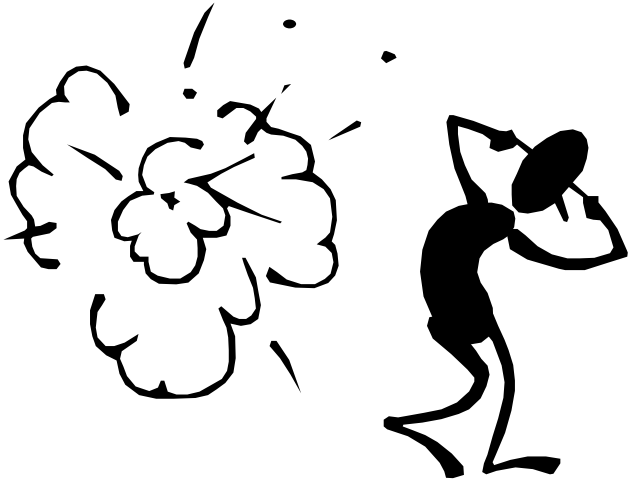
If an exception occurs on this line, the rest of the lines in the method are skipped and the program is terminated.

Terminated.

What is an exception?

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

Exception = Exceptional Event



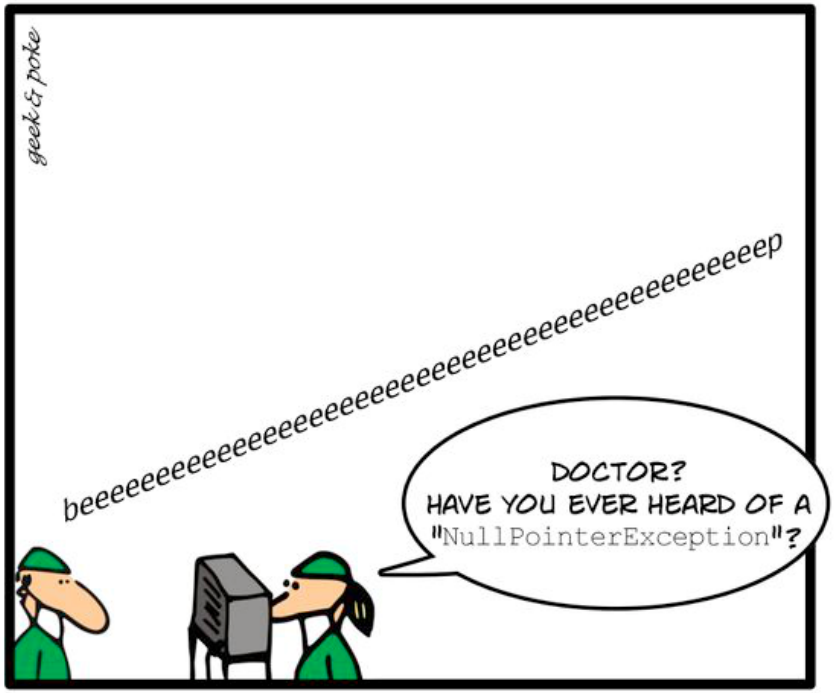
What is an exception?

- An exception is an abnormal condition that arises in a code sequence at **runtime**. For instance:
  - Dividing a number by zero
  - Accessing an element that is out of bounds of an array
  - Attempting to open a file which does not exist
- A Java exception is an object that describes an exceptional condition that has occurred in a piece of code
- When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error
- An exception can be caught to handle it or it can be passed on
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code

Exceptions

- A Method in Java throws exceptions to tell the calling code:

  "*Something bad happened. I failed.*"



RECENTLY IN THE OPERATING ROOM

What is an exception? (Example)

```
1- public class ExceptionExample {
2-   public static void main(String[] args) {
3-       int dividend = 5;
4-       int divisor = 0;
5-       int division = dividend / divisor;  // !!! Division by zero!
6-       System.out.println(" Result: " + division);
7-   }
8- }
```

Program "*crashes*" on the 5th line and the output is:

Exception in thread "main" java.lang.ArithmeticException: / by zero

at ExceptionExample.main(ExceptionExample.java:5)

Does the program really "crash"?

- Division by zero is an abnormal condition!
- Java run-time system cannot execute this condition normally
- Java run-time system creates an exception object for this condition and *throws* it
- This exception can be caught in order to overcome the abnormal condition and to make the program continue
- There is no exception handling code in the example program, so JVM terminates the program and displays what went wrong and where it was. Remember the output:

*Exception in thread "main" java.lang.ArithmeticException: **/ by zero***

*at ExceptionExample.main(**ExceptionExample.java:5**)*

What is exception handling?

- Exception mechanism gives the programmer a chance to do something against an abnormal condition.

- Exception handling is performing an action in response to an exception.

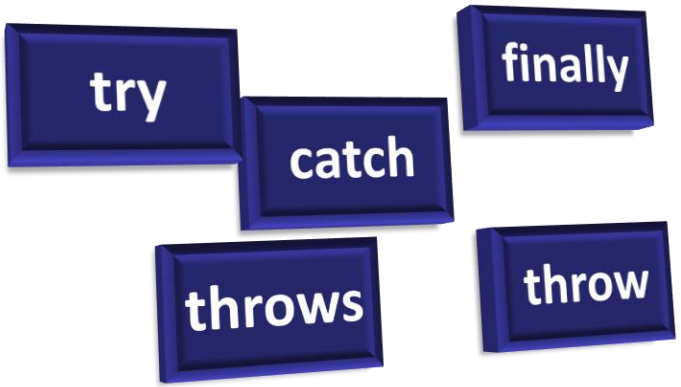
- This action may be:
  - Exiting the program
  - Retrying the action with or without alternative data
  - Displaying an error message and warning user to do something
  - ...

Keywords of Exception Handling

- There are five keywords in Java to deal with exceptions: **try, catch, throw, throws** and **finally.**

- **try**: Creates a block to monitor if any exception occurs.

- **catch**: Follows the try block and catches any exception which is thrown within it.



Let's *try* and *catch*

```
1-  public class ExceptionExample {
2-     public static void main(String[] args) {
3-       try {
4-           int dividend = 5;
5-           int divisor = 0;
6-           int division = dividend / divisor;  // !!! Division by zero!
7-           System.out.println(" Result: " + division);
8-       } catch (Exception e) {
9-           System.out.println ("Exception occurred and handled!" );
10-      }
11-    }
12- }
```

What happens when we *try* and *catch*?

- **int division = dividend / divisor;** statement causes an exception
- Java run-time system throws an exception object that includes data about the exception
- Execution stops at the 6th line, and a catch block is searched to handle the exception
- Exception is caught by the 8th line and execution continues by the 9th line
- Output of the program is:

  Exception occurred and handled!

Let's visualize it!

```
1-  public class ExceptionExample {
2-    public static void main(String[] args) {
3-      try {
4-        int dividend = 5;
5-        int divisor = 0;
6-        int division = dividend / divisor;
7-        System.out.println(" Result: " + division);
8-      } catch (Exception e) {
9-        System.out.println (" Exception occurred! " );
10-     }
11-   }
12- }
```

1. An exception is thrown by JVM

Exception object is created

e is a reference to the exception object

2. Execution stops at the exception line and diverges to the following catch block

try and catch statement

- The scope of a **catch** clause is restricted to those statements specified by the immediately preceding **try** statement.
- A **catch** statement cannot catch an exception thrown by another **try** statement.
- The statements that are protected by the **try** must be surrounded by curly braces.

Are there many exceptions in Java?

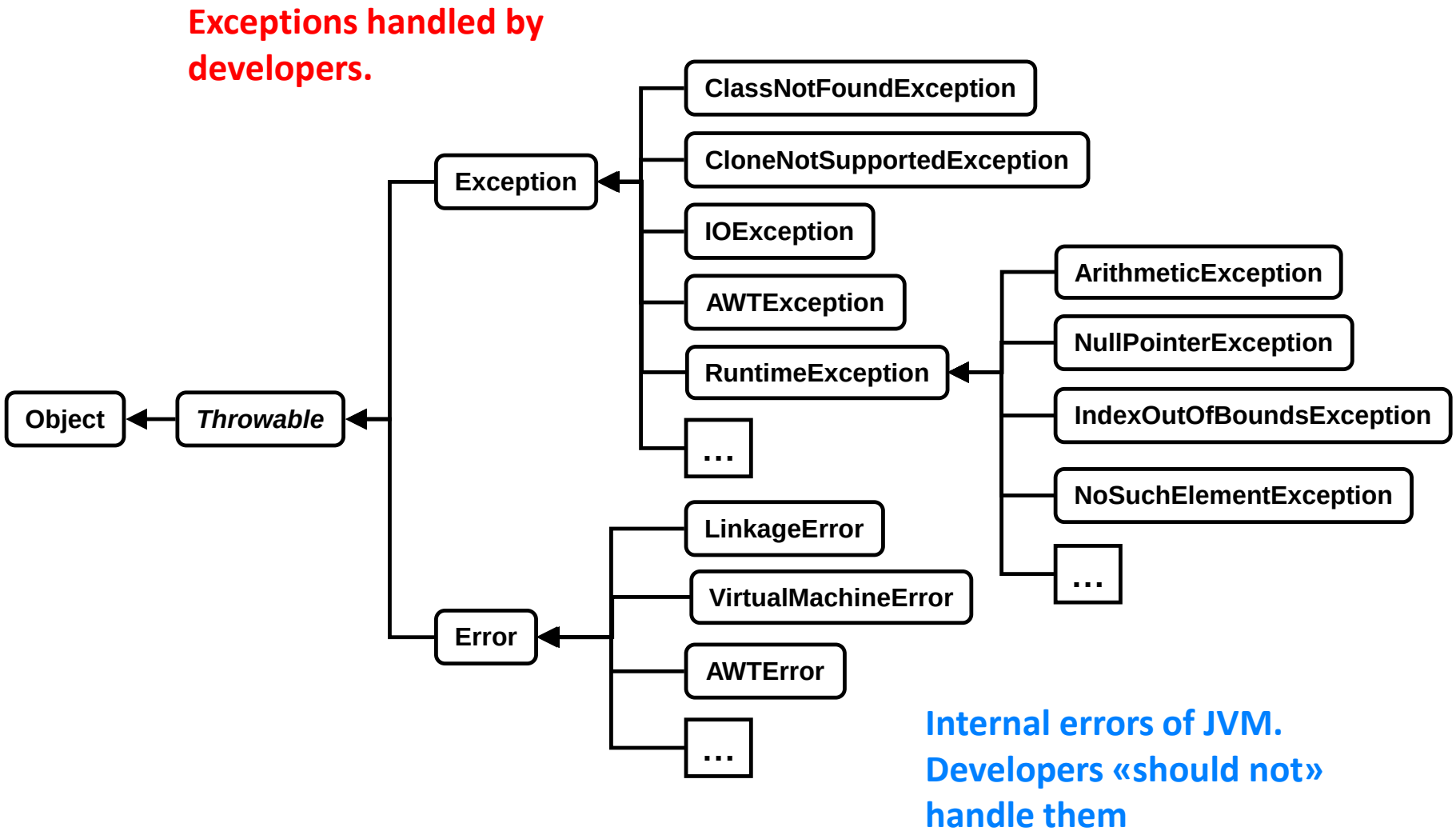
- Yes! Check the Java API Documentation at http://docs.oracle.com/javase/8/docs/api/

- `java.lang.Exception` is the base class of the exception hierarchy

- There are many direct and indirect subclasses of java.lang.Exception, for example
  - `java.lang.ArithmeticException`
  - `java.lang.ArrayIndexOutOfBoundsException`
  - `java.lang.NullPointerException`
  - `java.io.IOException`
  - `java.io.FileNotFoundException`

- We can also write custom exception classes

Hierarchy of Exception Classes in Java



Multiple catch clauses

- It is possible that more than one exception can be thrown in a code block.
  - We can use multiple **catch** clauses
- When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type *matches* that of the exception is executed.
  - Type matching means that the exception thrown must be an object of the same class or a sub-class of the declared class in the **catch** statement
- After one **catch** statement executes, the others are bypassed.

Multiple catch statement example

```
try {
    System.out.print("Give me an integer: ");
    int number = (new Scanner(System.in)).nextInt();
    System.out.println("10 / " + number + " is: " + (10 / number));
    int array[] = new int[]{1, 2, 3, 4, 5};
    System.out.println("array[" + number + "]: " + array[number]);
}
catch (ArithmeticException e) {
    System.out.println("Division by zero is not possible!");
}
catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Number is out of the array!");
}
```

ArithmeticException may occur

ArrayIndexOutOfBoundsException may occur

Multiple catch statement example

- 1st scenario: Assume that user enters value 2. What is the output of the program?

```
Give me an integer: 2
10 / 2 is: 5
array[2] is: 3
```

- 2nd scenario: Assume that user enters value 5. What is the output of the program?

```
Give me an integer: 5
10 / 5 is: 2
Number is out of the array!
```

- 3rd scenario: Assume that user enters value 0. What is the output of the program?

```
Give me an integer: 0
Division by zero is not possible!
```

Multiple catch clauses and inheritance

- If there is inheritance between the exception classes which are written in catch clauses;
  - Exception subclass must come before any of their superclasses
  - A catch statement that uses a superclass will catch exceptions of that type **plus any of its subclasses**. So, the subclass would never be reached if it comes after its superclass

```
catch (Exception e) {
}
catch (ArithmeticException e) {
}
```

Compile error! Second clause is unnecessary, because first clause will catch any exception!

```
catch (ArithmeticException e) {
}
catch (Exception e) {
}
```

It is OK now! Any exception other than an ArithmeticException will be caught by the second clause!

More on multiple catch clauses

- **Multiple catch clauses** give programmer the chance to take different actions for each exception

..., but a new catch clause for each possible exception will possibly make the code so complex

- **A single catch clause** with the java.lang.Exception will catch any exception thrown

..., but the programmer will not know which exception was thrown!

Confused about multiple catch clauses?

- Programmer decides on the details of the exception handling strategy
  - If it is just enough to know that something went wrong and the same action will be taken for all exceptions (for instance; displaying a message), then use a single catch clause with Exception!
  - If it is really necessary to know which exception occurs and different actions will be taken for each exception, then use multiple catch clauses!

Catching Exceptions

```
try {
       //Statements that may throw exceptions
}

catch (Exception1 exVar1) {
  //code to handle exceptions of type Exception1;
}

catch (Exception2 exVar2) {
  // code to handle exceptions of type Exception2;
}

...
catch (ExceptionN exVarN) {
  // code to handle exceptions of type exceptionN;
}

// statement after try-catch block
```

Nested try statements

- A **try** block can include other **try** block(s)

```
try {
    ...
    try {
        ...
    } catch (Exception e) {
        ...
    }
    ...
} catch (Exception e) {
    ...
}
```

Nested try statements

- A try block can call a method which has a try block in it.

```
try {
    ...
    method();
} catch (Exception e) {
    ...
}
```

```
void method() {
    try {
        ...
    } catch (Exception e) {
        ...
    }
}
```

Nested try statements

An exception is thrown in

```
main method {
  ...
  try {
    ...
    invoke method1;
    statement1;
  }
  catch (Exception1 ex1) {
    //Process ex1;
  }
  statement2;
}
```

```
method1 {
  ...
  try {
    ...
    invoke method2;
    statement3;
  }
  catch (Exception2 ex2) {
    //Process ex2;
  }
  statement4;
}
```

```
method2 {
  ...
  try {
    ...
    invoke method3;
    statement5;
  }
  catch (Exception3 ex3) {
    //Process ex3;
  }
  statement6;
}
```

Nested try statements

- When an exception occurs inside a **try** block;
  - If the **try** block does not have a matching catch, then the outer **try** statement's catch clauses are inspected for a match
  - If a matching catch is found, that catch block is executed
  - If no matching catch exists, execution flow continues to find a matching catch by inspecting the outer try statements
  - If a matching catch cannot be found, the exception will be caught by JVM's exception handler.

- Caution! Execution flow never returns to the line that exception was thrown.
  - This means, an exception is caught and catch block is executed, the flow will continue with the lines following this catch block

Let’s clarify it on various scenarios

```
try {
   statement1;
   try {
            statement2;
   } catch (Exception1 e) {
            statement3;
   } catch (Exception2 e) {
            statement4;
   }
   try {
            statement5;
   } catch (Exception3 e) {
            statement6;
   }
   statement7;
} catch (Exception3 e) {
   statement8;
}
statement9;
```

Information: Exception1 and Exception2 are subclasses of Exception3

Question: Which statements are executed if

1- statement1 throws Exception1

2- statement2 throws Exception1

3- statement2 throws Exception3

4- statement2 throws Exception1 and statement3 throws Exception2

Scenario: statement1 throws Exception1

```
try {
    statement1;
    try {
            statement2;
    } catch (Exception1 e) {
            statement3;
    } catch (Exception2 e) {
            statement4;
    }
    try {
            statement5;
    } catch (Exception3 e) {
            statement6;
    }
    statement7;
} catch (Exception3 e) {
    statement8;
}
statement9;
```

Step1: Exception is thrown

Exception1

Step2: catch clauses of the try block are inspected for a matching catch statement. Exception3 is super class of Exception1, so it matches.

Step3: statement8 is executed, exception is handled and execution flow will continue bypassing the following catch clauses

Step4: statement9 is executed

Scenario: statement2 throws Exception1

```
try {
   statement1;
   try {
            statement2;
   } catch (Exception1 e) {
            statement3;
   } catch (Exception2 e) {
            statement4;
   }
   try {
            statement5;
   } catch (Exception3 e) {
            statement6;
   }
   statement7;
} catch (Exception3 e) {
   statement8;
}
statement9;
```

Step1: Exception is thrown

Exception1

Step2: catch clauses of the try block are inspected for a matching catch statement. First clause catches the exception

Step3: statement3 is executed, exception is handled

Step4: execution flow will continue bypassing the following catch clauses. statement5 is executed.

Step5: Assuming no exception is thrown by statement5, program continues with statement7 and statement9.

Scenario: statement2 throws Exception3

```
try {
   statement1;
   try {
            statement2;
   } catch (Exception1 e) {
            statement3;
   } catch (Exception2 e) {
            statement4;
   }
   try {
            statement5;
   } catch (Exception3 e) {
            statement6;
   }
   statement7;
} catch (Exception3 e) {
   statement8;
}
statement9;
```

Step1: Exception is thrown

Exception3

Step2: catch clauses of the try block are inspected for a matching catch statement. None of these catch clauses match Exception3

Step3: Catch clauses of the outer try statement are inspected for a matching catch. Exception3 is caught and statement8 is executed

Step4: statement9 is executed

Scenario: statement2 throws Exception1 and statement3 throws Exception2

```
try {
   statement1;
   try {
            statement2;
   } catch (Exception1 e) {
            statement3;
   } catch (Exception2 e) {
            statement4;
   }
   try {
            statement5;
   } catch (Exception3 e) {
            statement6;
   }
   statement7;
} catch (Exception3 e) {
   statement8;
}
statement9;
```

Step1: Exception is thrown

Exception1

Step2: Exception is caught and statement3 is executed.

Step3: statement3 throws a new exception

Exception2

Step4: Catch clauses of the outer try statement are inspected for a matching catch. Exception2 is caught and statement8 is executed

Step5: statement9 is executed

finally

- **finally** creates a block of code that will be executed after a **try/catch** block has completed and before the following **try/catch** block
- **finally** block is executed whether or not exception is thrown
- **finally** block is executed whether or not exception is caught
- It is used to gurantee that a code block will be executed in any condition.

finally

- Use *finally* clause for code that must be executed "no matter what"

```
try {
       //Statements that may throw exceptions
}

catch (Exception1 exVar1) {
  //code to handle exceptions of type Exception1;
}

catch (Exception2 exVar2) {
  // code to handle exceptions of type Exception2;
}
...
catch (ExceptionN exVar3) {
  // code to handle exceptions of type exceptionN;
}

finally { // optional
   // code executed whether there is an exception or not
}
```

Let's clarify it on various scenarios

```
try {
        statement1;
} catch (Exception1 e) {
        statement2;
} catch (Exception2 e) {
        statement3;
} finally {
   statement4;
}
statement5;
```

Question: Which statements are executed if

1- no exception occurs

2- statement1 throws Exception1

3- statement1 throws Exception3

Scenario: no exception occurs

```
try {
        statement1;
} catch (Exception1 e) {
        statement2;
} catch (Exception2 e) {
        statement3;
} finally {
   statement4;
}
statement5;
```

Step1: statement1 is executed

Step2: finally block is executed, statement4 is executed

Step3: statement5 is executed

Scenario: statement1 throws Exception1

```
try {
        statement1;
} catch (Exception1 e) {
        statement2;
} catch (Exception2 e) {
        statement3;
} finally {
   statement4;
}
statement5;
```

Step1: Exception is thrown

Exception1

Step2: catch clauses of the try block are inspected for a matching catch statement. Exception1 is caught and statement2 is executed.

Step3: finally block is executed, statement4 is executed.

Step4: statement5 is executed

Scenario: statement1 throws Exception3

```
try {
        statement1;
} catch (Exception1 e) {
        statement2;
} catch (Exception2 e) {
        statement3;
} finally {
   statement4;
}
statement5;
```

Step1: Exception is thrown

Exception3

Step2: catch clauses of the try block are inspected for a matching catch statement. There is no matching catch. finally is executed before inspecting the outer block. statement4 is executed.

Step3: statement5 is not executed, a matching catch will be inspected at outer block(s)

throw

- Developer can *throw* exceptions. Keyword **throw** is used for this purpose:

  throw *ThrowableObject*

- *ThrowableObject* is the object to be thrown. It must directly or indirectly extend the class **java.lang.Throwable**

- Developer can create a new object of an exception class, or rethrow the caught exception

Throwing and rethrowing example

```
import java.util.Scanner;

public class ThrowingExample {
    public static void main(String[] args) {
        System.out.print("Give me an integer: ");
        int number = new Scanner(System.in).nextInt();
        try {
            if (number < 0)
                throw new RuntimeException();
            System.out.println("Thank you.");
        } catch (Exception e) {
            System.out.println("Number is less than 0!");
            throw e;
        }
    }
}
```

Keyword **throw** is used to throw an exception.

e is already reference of an exception object. It can also be used to throw (rethrow) that exception

Coding custom exception classes

- Developer can also code custom exception classes to manage abnormal conditions in his program
- If a class extends Throwable, that class can be thrown
- We usually prefer to extend class Exception or RuntimeException (difference of these two will be explained)
- Extending an exception class and coding necessary constructors is enough to create a custom exception class

Custom exception example

```
public class LessThanZeroException extends Exception {
        public LessThanZeroException() {
        }
        public LessThanZeroException(String message) {
                super(message);
        }
}
import java.util.Scanner;
public class ThrowingExample {
    public static void main(String[] args) {
        System.out.print("Give me an integer: ");
        int number = new Scanner(System.in).nextInt();
        try {
            if (number < 0)
                throw new LessThanZeroException();
            System.out.println("Thank you.");
        } catch (LessThanZeroException e) {
            System.out.println("Number is less than 0!");
        }
    }
}
```

Getting data from the exception object

- **Throwable** overrides the **toString()** method (defined by class **Object**) so that it returns a string containing a description of the exception

Example:

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

Output:

```
Exception is: java.lang.ArithmeticException: / by zero
```

Getting data from the exception object

- **Throwable** class also has useful methods. One of these methods is the **getMessage()** method
- The message that is put in the exception (via the constructor with String parameter) can be taken by **getMessage()** method

Example:

```
catch(ArithmeticException e) {
        System.out.println("Problem is: " + e.getMessage());
}
```

Output:

Problem is: / by zero

Getting data from the exception object

- Another method is the **printStackTrace()** method
- This method is used to see what happened and where

Example:

```
catch(ArithmeticException e) {
        e.printStackTrace();
}
```

Output:

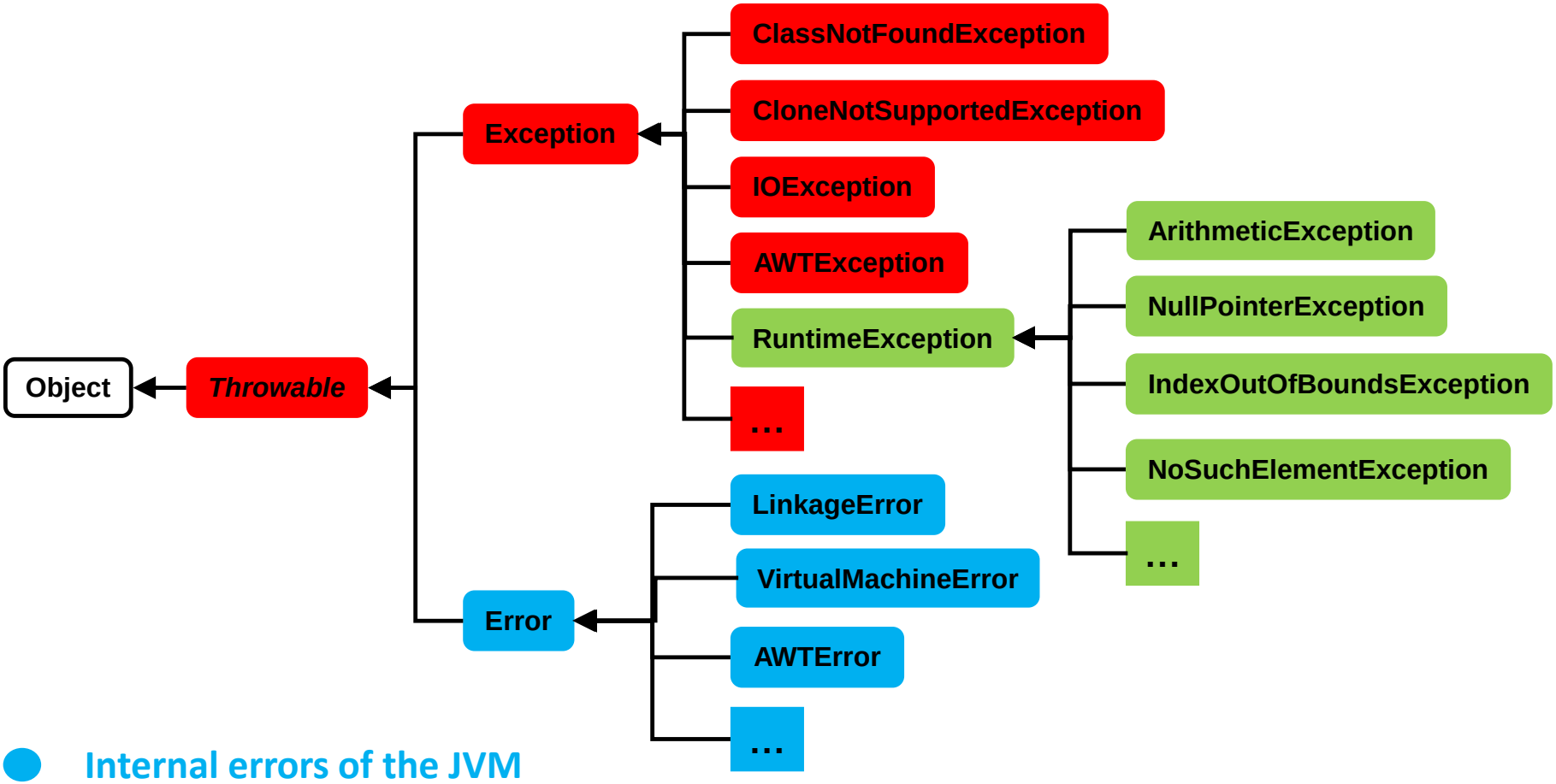
```
java.lang.ArithmeticException: / by zero
        at ExceptionExample.main(ExceptionExample.java:6)
```

This output means:
A java.lang.ArithmeticException occurred at 6th line of the main method of the ExceptionExample class

Did you recognize that... ?

- The output of the **printStackTrace()** method is very similar to the output you have seen before...
- You have seen it when your programs crashed!
- When an exception is not caught by the program, JVM catches it and prints the stack trace to the console.
- This output is very helpful to find the errors in the program

Checked and Unchecked Exceptions



- Internal errors of the JVM
- Unchecked exceptions
- Checked exceptions

What does *Checked Exception* mean?

- If a method will possibly throw an exception, compiler *checks* the type of the exception
- if the exception is a checked exception, compiler forces the developer to do one of these:
  - write a matching catch statement for that exception
  - declare that the method will possibly throw that exception

Handling Checked Exceptions

- Java forces you to deal with checked exceptions.

- Two possible ways to deal:

```
void p1() {
  try {
    riskyMethod();
  }
  catch (IOException ex) {
    ...
  }
}
```

(a)

```
void p1() throws IOException {

  riskyMethod();

}
```

(b)

throws

- Keyword **throws** is used to declare that a method is capable of throwing exception(s)
- Callers of the method can guard themselves against that exception(s)

Examples:

```
public void m1() throws Exception1 {
}

public void m2() throws Exception1, Exception2, Exception3 {
}
```

CheckedExceptionExample1

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class CheckedExceptionExample1 {
    public static void main(String[] args) {
        System.out.println("Line: " + readALine1("input.txt"));
    }

    public static String readALine1(String filename) {
        try {
            BufferedReader inputFile = new BufferedReader(new FileReader("a.txt"));
            String line = inputFile.readLine();
            inputFile.close();
            return line;
        } catch (IOException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

FileNotFoundException may be thrown here

IOException may be thrown here

IOException is super class of FileNotFoundException

CheckedExceptionExample2

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class CheckedExceptionExample2 {
    public static void main(String[] args) {
        try {
            System.out.println("Line: " + readALine2("input.txt"));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static String readALine2(String filename) throws IOException {
        BufferedReader inputFile = new BufferedReader(new FileReader("a.txt"));
        String line = inputFile.readLine();
        inputFile.close();
        return line;
    }
}
```

IOException is superclass of FileNotFoundException. No need to declare both.

FileNotFoundException may be thrown here

IOException may be thrown here

What does *Unchecked Exception* mean?

- If a code block has the possibility of throwing an unchecked exception, compiler does not force the developer for anything. It is up to the developer to do one of these:
  - to handle the exception
  - let the program crash

Does a developer let his program crash?

- Unchecked exceptions are usually results of the developer's mistakes.
  - For example, if a reference may normally be null, then it is developer's responsibility to check if it is null or not. NullPointerException should not occur in this scenario!
  - Letting program crash at the development phase will make the developer find such errors and potential bugs.

Summary

- Exceptions are used to take actions against abnormal conditions
- Exceptions are objects which are *thrown* by JVM or the developer's code
- There are many exception classes in standard java library, and custom exception classes can be coded
- Exception handling is *catching* an exception and taking an action against it
- Keywords **try**, **catch**, and **finally** are used for exception handling
- Exceptions are classified as unchecked (RuntimeException class and its subclasses), or checked (Throwable class and its subclasses, except Error and RuntimeException)
- If a method has the capability of throwing a checked exception, it must either handle the exception (with try/catch blocks), or declare it with keyword **throws**

References

- Ganesh Wisvanathan, CIS3023: Programming Fundamentals for CIS Majors II, University of Florida