COMPUTER PROGRAMMING
Unit 3
Elements of Object Oriented Programming. Inheritance and Exception Handling

Jon Ander Gómez Adrián

`jon@dsic.upv.es`

Departament de Sistemes Informàtics i Computaciò
Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

25 de enero de 2018

UNIVERSITAT POLITÈCNICA DE VALÈNCIA DSIC Escola Tècnica Superior d'Enginyeria **Informàtica** ets**inf**

Contents

Contents

- "Empezar a programar usando Java" - **Chapters 14 and 15**
  IIP and PRG teachers
  In *PoliformaT* of PRG.

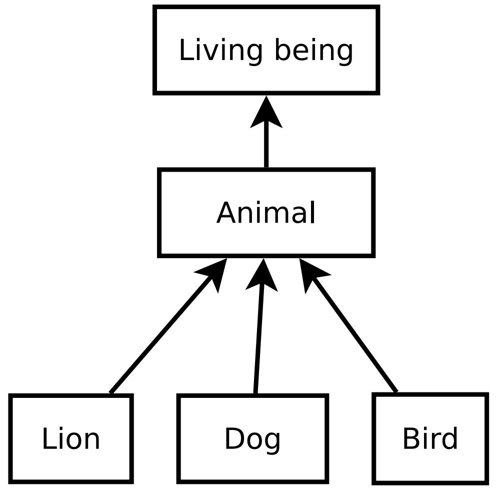
- "Introduction to Programming Using Java, Sixth Edition"
  David J. Eck
  `http://math.hws.edu/javanotes/`
  Chapter 3 (3.7), Chapter 5 (5.5 & 5.6) & Chapter 8 (8.3)

- "Estructuras de datos en Java: compatible con Java 2"
  Mark Allen Weiss
  Ed. Addison Wesley, 2000 - 2006
  Chapter 2 (2.5) & Chapter 4 (4.1, 4.2 i 4.3)

- "The Java^{TM} Tutorials Oracle"
  `http://download.oracle.com/javase/tutorial/`
  Trail: Learning the Java Language. Lesson: Interfaces and Inheritance
  Trail: Essential Java Classes. Lesson: Exceptions.

Inheritance

- Inheritance is a mechanism provided by object oriented programming languages for reusing the designs of existing classes when defining new ones.
- Inheritance allows to model relations among classes for defining a hierarchy of classes.
- Then, Java classes can use variables and methods defined into classes located on an upper level inside the hierarchy.
- A good example is the relation between different species of life. Or the hierarchy started from Java predefined class `Number`, available in the Java API.



Inheritance

Given the class `Person` define the class `Student` with three attributes: name, age, and ects (number of credits for which the student is enrolled).

```
public class Person {
    private String name;
    private int    age;
    public Person( String name, int age ) {
        this.name = name;
        this.age = age;
    }
    public String getName() { return name; }
    public int getAge() { return age; }
    public String toString() {
        return "Name: " + name + " Age: " + age;
    }
}
```

Possible options:

- Not Appropriate: to ignore the class `Person` already defined and to design the class `Student` from scratch.
- Appropriate: using inheritance for defining the class `Student` from the class `Person`.

Inheritance

```
public class Person {
    private String name;
    private int    age;
    public Person( String name, int age ) {
        this.name = name;
        this.age = age;
    }
    public String getName() { return name; }
    public int getAge() { return age; }
    public String toString() {
        return "Name: " + name + " Age: " + age;
    }
}
```

```
public class Student extends Person {
    private int    ects;
    public Student( String name, int age ) {
        super( name, age );
        this.ects = 60;
    }
    public int getECTS() { return ects; }
    public String toString() { return super.toString() + " ECTS: " + ects; }
}
```

Inheritance

```
public class TestStudent
{
    public static void main( String args[] )
    {
        Student s = new Student( "John Smith", 19 );

        System.out.printf( "Person: %s\n", s.toString() );
        System.out.printf( "%s is enrolled in %d ECTS.\n", s.getName(), s.getECTS() );
    }
}
```

- We can use all public methods defined in the class `Person` from objects of the class `Student`. Obviously, private methods of a class can only be invoked from other methods of the same class.
- Objects of the class `Student` have the attributes and the methods defined in class `Person`, in addition to the attributes and methods defined in the class `Student`.
- Objects of class `Student` have no direct access to private attributes defined in the class `Person`.

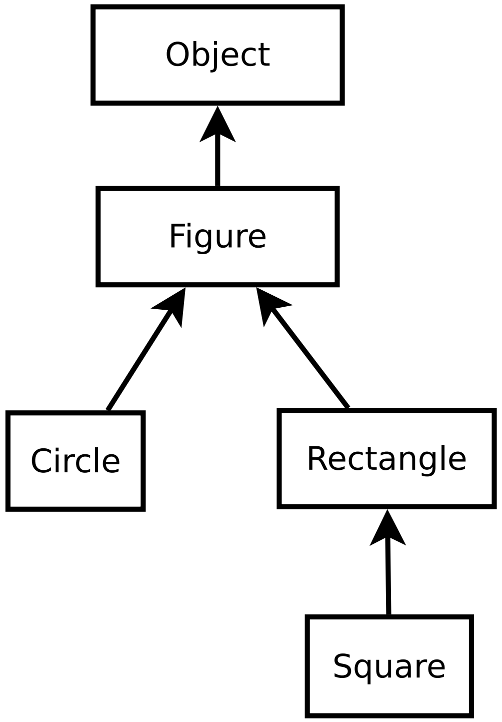
Hierarchy of classes and subclasses

- If class `B` extends class `A`, then we can say . . .
  - `B` is a sub-class of `A`
  - `A` is the super-class of `B`
  - `B` is a derived class and is a variation of class `A`
- The relation is transitive, if `C` is a sub-class of `B` then `C` is also a sub-class of `A`.
- Once class `B` is defined as an extension from class `A`, class `B` is a new class. Changes in `B` do not imply changes in `A`.
- Objects of class `B` can also play the role of objects of class `A`.

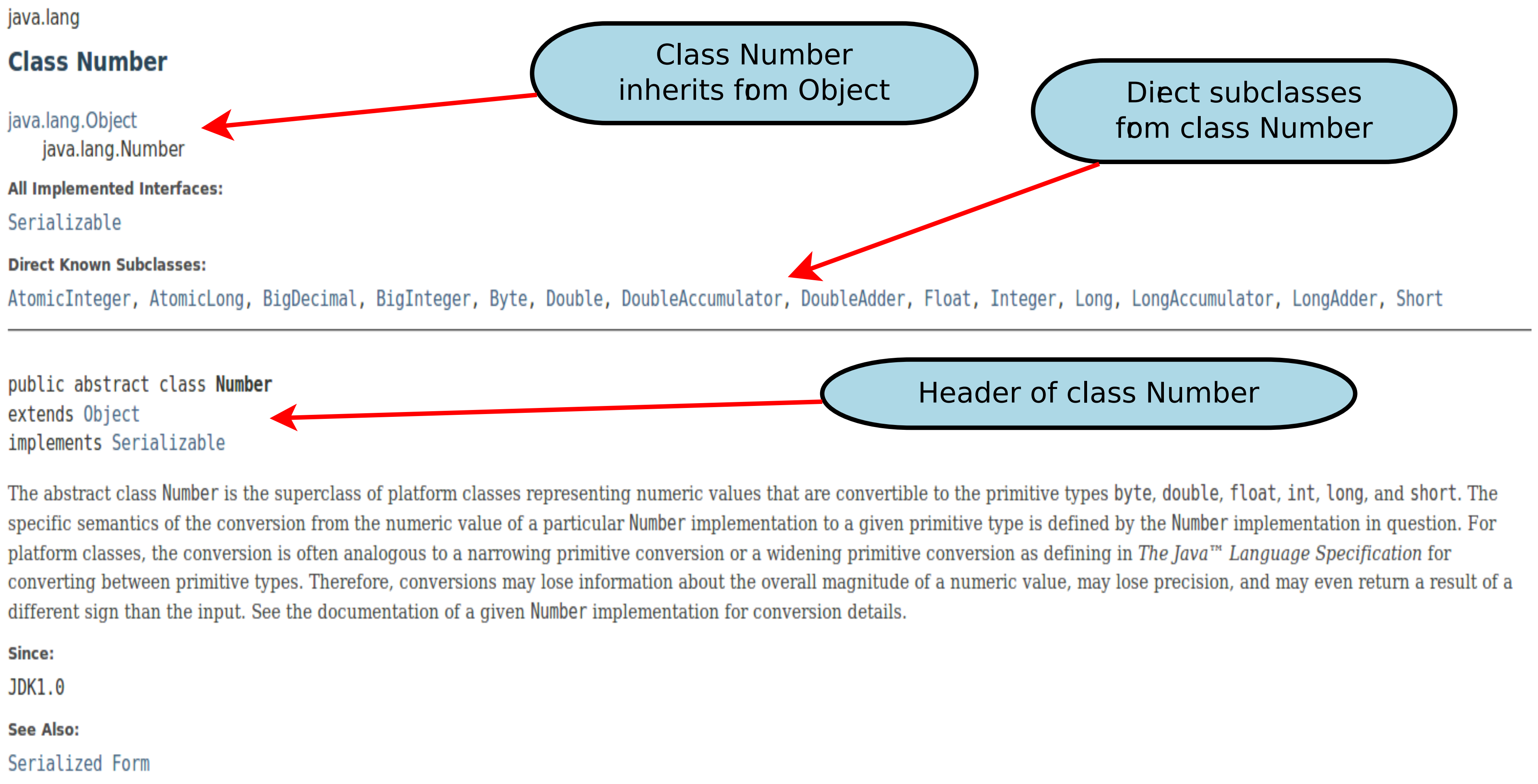
Java doesn't support multiple inheritance. Each class can only extend one class.

Hierarchy of classes and subclasses

- Any Java class is by default a subclass of the predefined class `Object`.
- The class `Object` is the root of the Java hierarchy of classes and subclasses.
- Since any class is derived directly or indirectly from `Object` class, the basic methods defined in the `Object` class can be invoked with respect to any created object:
  - `public String toString()`
  - `public boolean equals( Object other )`
  - `protected Object clone()`
- In the example of the picture:
  - A `Figure` is an `Object`.
  - A `Circle` is a `Figure`.
  - A `Rectangle` is a `Figure`.
  - A `Square` is a `Rectangle`.



Inheritance in the Java API documentation



java.lang

Class Number

java.lang.Object
    java.lang.Number

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

AtomicInteger, AtomicLong, BigDecimal, BigInteger, Byte, Double, DoubleAccumulator, DoubleAdder, Float, Integer, Long, LongAccumulator, LongAdder, Short

```
public abstract class Number
extends Object
implements Serializable
```

The abstract class `Number` is the superclass of platform classes representing numeric values that are convertible to the primitive types `byte`, `double`, `float`, `int`, `long`, and `short`. The specific semantics of the conversion from the numeric value of a particular `Number` implementation to a given primitive type is defined by the `Number` implementation in question. For platform classes, the conversion is often analogous to a narrowing primitive conversion or a widening primitive conversion as defining in *The Java™ Language Specification* for converting between primitive types. Therefore, conversions may lose information about the overall magnitude of a numeric value, may lose precision, and may even return a result of a different sign than the input. See the documentation of a given `Number` implementation for conversion details.

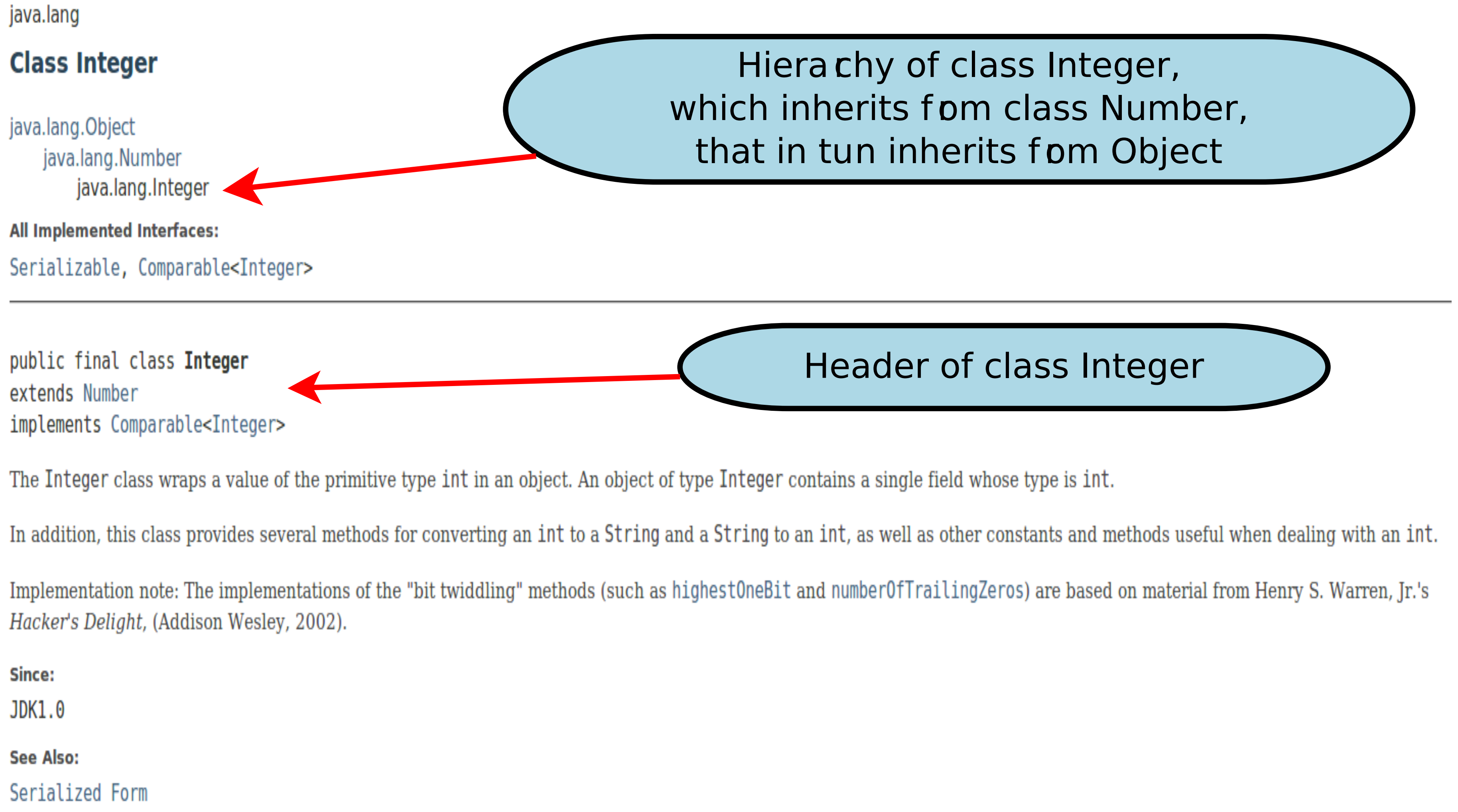
Since:

JDK1.0

See Also:

Serialized Form

Inheritance in the Java API documentation


Hiera chy of class Integer,
which inherits f om class Number,
that in tun inherits f om Object
Header of class Integer

java.lang

Class Integer

java.lang.Object
    java.lang.Number
        java.lang.Integer

All Implemented Interfaces:

`Serializable, Comparable<Integer>`

```
public final class Integer
extends Number
implements Comparable<Integer>
```

The `Integer` class wraps a value of the primitive type `int` in an object. An object of type `Integer` contains a single field whose type is `int`.

In addition, this class provides several methods for converting an `int` to a `String` and a `String` to an `int`, as well as other constants and methods useful when dealing with an `int`.

Implementation note: The implementations of the "bit twiddling" methods (such as `highestOneBit` and `numberOfTrailingZeros`) are based on material from Henry S. Warren, Jr.'s *Hacker's Delight*, (Addison Wesley, 2002).

Since:

`JDK1.0`

See Also:

`Serialized Form`

Escola Tècnica Superior d'Enginyeria **Informàtica**

Methods overriding

If class `B` is a subclass of class `A`, then

- Any non private method defined in `A` can be overridden in `B`.
- Complete overriding: when the body of the new method doesn't use the overridden method.
- Partial overriding: when the body of the new method do use the overridden method.
  `super` is used for invoking the method defined in the superclass.

Object	`public String toString() {`
	`    // Default implementation`
	`}`

Person	`public String toString() {`
	`    return "Name: "+ name + .Age: "+ age;`
	`}`

Student	`public String toString() {`
	`    return super.toString() + .ECTS: "+ ects;`
	`}`

Class `Person` overrides *completely* `toString()` method from class `Object`, but class `Student` overrides *partially* `toString()` method from class `Person`.

Methods overriding

```
public class TestStudent2
{
    public static void main( String args[] )
    {
        Student s = new Student( "John Smith", 19 );
        Person  p = new Person(  "John Smith", 19 );

        System.out.println( "Person.: " + p.toString() );
        System.out.println( "Student: " + s.toString() );
    }
}
```

- The output of this little program is:

```
Person.: Name: John Smith  Age: 19
Student: Name: John Smith  Age: 19  ECTS: 60
```

- It is not necessary to write `toString()` explicitly when the context in that objects are used there isn't ambiguity. Java calls implicitly to method `toString()`.

Exceptions handling in Java

What happens when it occurs an unexpected error, or an exception which was not planned?

For example:

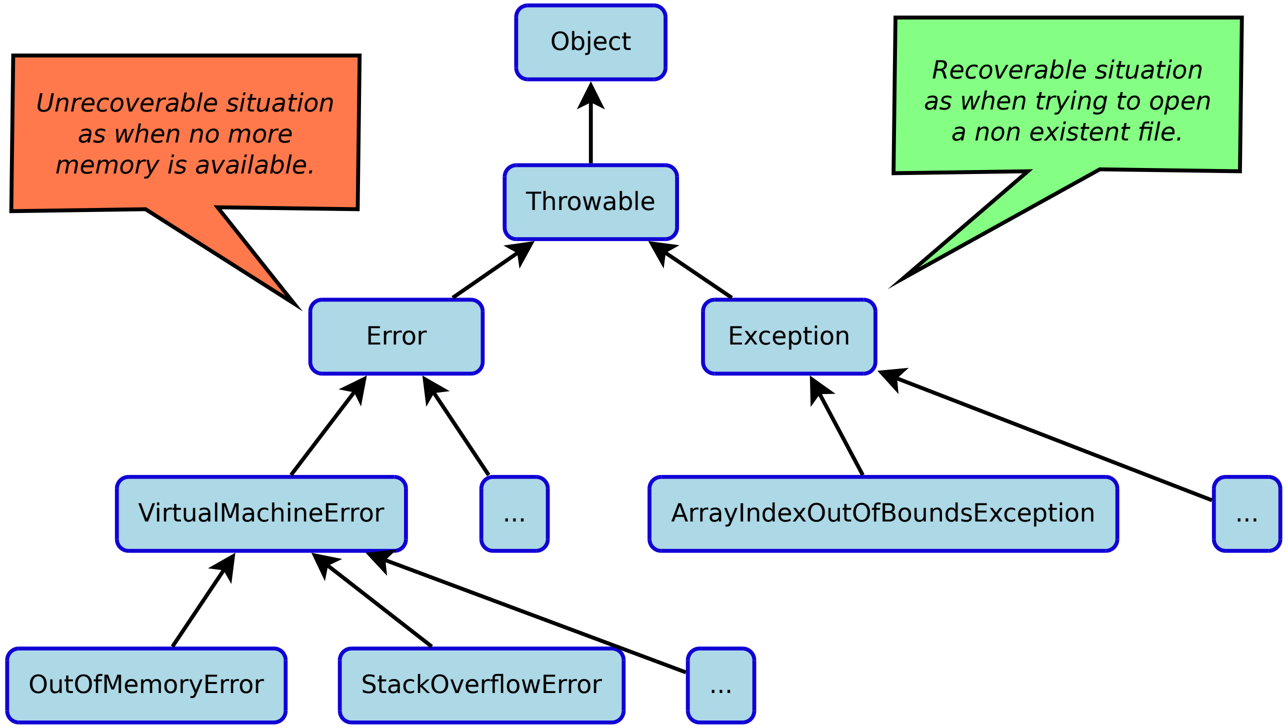
- Try to open an non existent file.
- User types a letter instead of a number.
- Invoking a method with respect to a `null` reference.
- Accessing non existent positions in arrays.
- Try to read from a file after reaching the end-of-file.

Some of these errors are semantic errors, but other ones are not due to the programmer.

Must these errors abort the execution of the program in all cases?

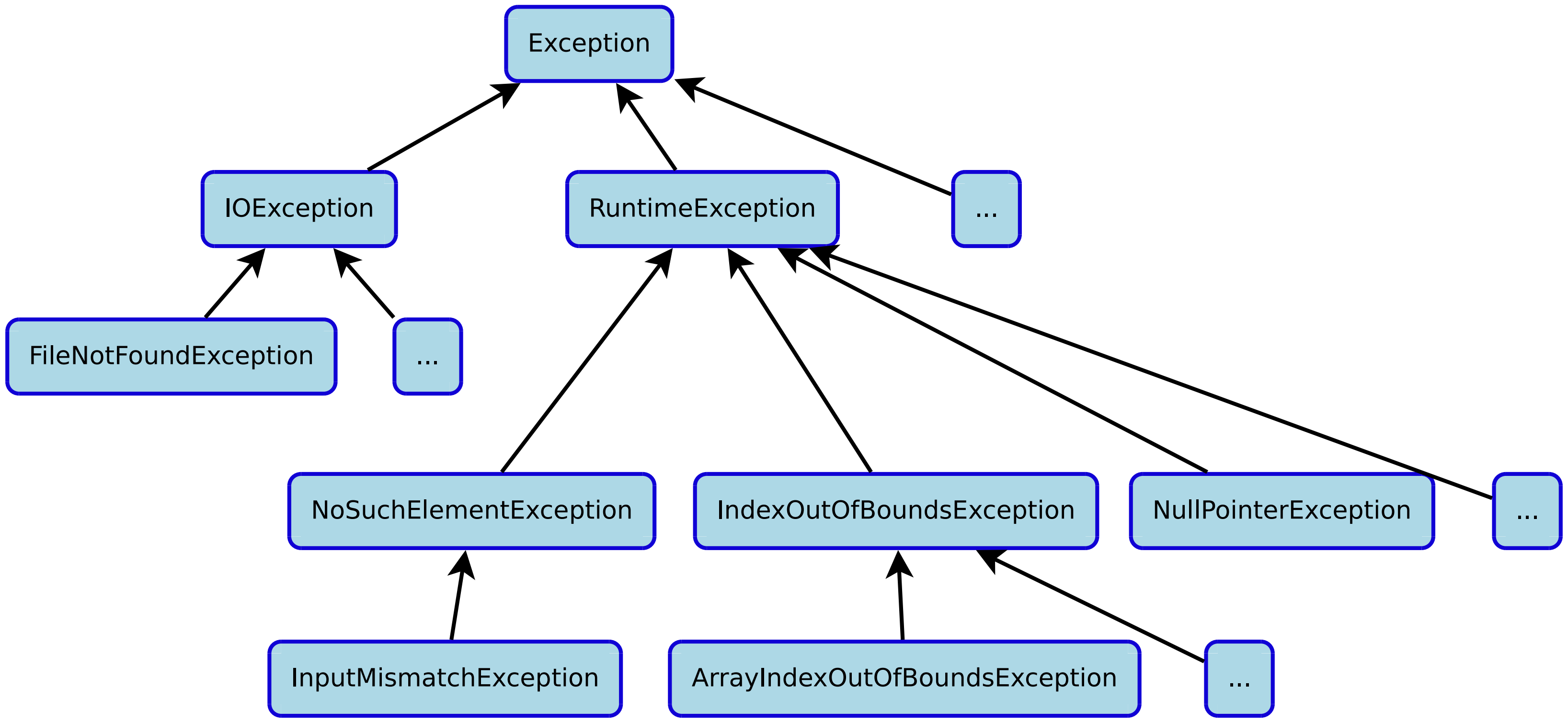
Exceptions handling in Java

Java has a hierarchy for representing execution errors, all these classes are subclasses from `Throwable`, and there exist both families recoverable and unrecoverable situations.


Object
Throwable
Error
Exception
VirtualMachineError
...
ArrayIndexOutOfBoundsException
...
OutOfMemoryError
StackOverflowError
...
Unrecoverable situation as when no more memory is available.
Recoverable situation as when trying to open a non existent file.

Exceptions handling in Java

Part of Java hierarchy of exceptions



Exceptions handling in Java

This unit is focused on class `Exception` and its derived subclasses, in particular:

- `IOException` for those exceptions related with I/O operations, such as `FileNotFoundException`
- `RuntimeException` for those exceptions related with programming errors, such as `ArrayIndexOutOfBoundsException`

Exceptions handling in Java

User defined exceptions

- User can define exceptions for taking into account foreseen errors in his computer-programs.
- For example, an application with a menu for selecting among five options. User can define a class derived from `Exception` as follows:

```
public class NonExpectedOptionException
    extends Exception
{
    public NonExpectedOptionException( String msg )
    {
        super( msg );
    }
}
```

- User defined exceptions are subclasses of class `Exception` and can include a message for describing the situation that leads to the error.

Exceptions handling in Java

What can we do once an exception has been thrown?

Exceptions can be *thrown*, *propagated* or *caught*.

An issue to be taken into account: all exceptions derived from `Exception` are considered checked exceptions, except the subfamily derived from `RuntimeException`.

`RuntimeException` and all its derived subclasses are considered unchecked exceptions.

Checked exceptions must be propagated or caught, otherwise Java compiler will show a compilation error. Unchecked exceptions can be ignored in the code, if they are thrown the program will abort.

Exceptions handling in Java

What can we do once an exception has been thrown?

An exception is thrown when the conditions that cause it are fulfilled. For example, when the program tries to open a non existent file. However, exceptions can be created and thrown.

```
Scanner input = new Scanner( System.in );

System.out.print( "Minutes? " );

int minutes = input.nextInt();

if ( minutes < 0 || minutes >= 60 ) {
    throw new InputMismatchException( "Out of range!" );
}
```

Exceptions handling in Java

- When an exception is thrown inside a method, the programmer can decide if this exception will be propagated or caught. Additionally the unchecked exceptions can be ignored.
- If the programmer decides not to catch an exception the default behaviour is to abort the program.
- Otherwise the `try-catch-finally` mechanism can be used.
  In this case the programmer can write the appropriate code for achieving that the program continues despite the values of some variables (or other conditions).
- Here you have an example of how some exceptions are propagated.

```
...
public void method1( <list of arguments> )
    throws InputMismatchException, IOException
{
}
...
```

Exceptions handling in Java

```
import java.util.*;
import java.io.*;
public class ExceptionHandling
{
    static Scanner input = new Scanner( System.in );

    public static void main( String args[] )
    {
        try {
            int minutes = getMinutes();
        }
        catch( InputMismatchException ime ) { /* Here the code. */ }
        catch( IOException ioe ) { /* Here the code. */ }
        catch( Exception e ) { /* Here the code. */ }
        finally { /* Code that will be executed always. */ }
    }
    public static int getMinutes() throws InputMismatchException, IOException
    {
        System.out.print( "Minutes? " );
        int minutes = input.nextInt();
        if ( minutes < 0 || minutes >= 60 )
            throw new InputMismatchException( "Minutes out of range!" );

        return minutes;
    }
}
```

Exceptions handling in Java

try segment of code containing the instructions where several kinds of exceptions can be thrown.

- If an exception is thrown, the execution thread jumps to the `catch` block corresponding to the thrown exception or one of its superclasses.
- If no exception is thrown, the `try` block is executed normally.

catch segment of code to be executed when an exception of a compatible type has been thrown.

- Must exist at least one `catch` block after each `try` block if there is not `finally` block.
- Each `catch` block is associated to an exception, i.e. the subtree of exceptions rooted in the exception specified as parameter of the `catch`.
- At least one `catch` for each possible exception.

finally segment of code whose instructions will be executed always.

- This is optional. It is useful when no checked exceptions have been thrown or when unrecoverable error conditions occur.
- The sequence of instructions in the `finally` block can be used to close opened files before the program aborts. In general for saving buffered data.

Exceptions handling in Java

- As shown in the last example, Java provides programmers with a mechanism for propagating exceptions, i.e., the exception is not caught inside the same method where it can be produced, but it can be caught in a method that calls the current one directly or indirectly.
- The syntax of this mechanism is to list the names of propagated exceptions after the reserved word `throws`.
- In the case of propagated exceptions, the execution of the method aborts when the exception occurs, but the program can continue if the exception is caught properly in one of the methods in the call stack pending of its execution ends.
- Remember! If the exception is checked then it must be propagated or caught.

Exceptions handling in Java

```
import java.util.*;
import java.io.*;
public class ExceptionHandling2
{
    static Scanner input = new Scanner( System.in );
    public static void main( String args[] )
    {
        int A[] = { 6, 7, 8, 9, 10, 11, 1, 3, 2, 4 }; boolean valueOK; int index=0;
        do {
            valueOK=true;
            try {
                System.out.print( "Position in the array? " ); index = input.nextInt();
                showValue( A, index );
            }
            catch( ArrayIndexOutOfBoundsException aioobe ) {
                System.out.printf( "ERROR: %d is not a valid index!\n", index );
                valueOK=false;
            }
        } while( !valueOK );
    }
    public static void showValue( int A[], int i ) throws ArrayIndexOutOfBoundsException
    {
        System.out.println( A[i] );
    }
}
```

Exceptions handling in Java

- Thanks to the `throws` clause any method can propagate several types of exceptions. Therefore, handling exceptions by means of a “catch” block can be programmed in other methods, instead of inside the method where the exceptions can occur.
- The `throws` clause can be followed by a list of class names representing different exceptions.
- Watch out and do not confuse the `throws` clause with the instruction `throw` for throwing exceptions.
- Checked exceptions must be propagated or caught.
- The flow control instructions `try--catch--finally` can be nested as other flow control instructions (`while/for`, `if-else` or `switch`).