

# Java Support Classes

Videogames Technology  
Asignatura transversal

Departamento de Automática

## Objectives

- Theoretical and practical understanding of exceptions
- Introduce elements of the Java API: Streams and JCF
- Review the most important data structures

## Bibliography

- The Java™ Tutorials. Oracle. ([Link](#))
- Collections Framework Overview. Oracle. ([Link](#))

# Table of Contents

## 1. Exceptions

- Exception definition
- try-catch
- Exceptions thrown by a method

## 2. Basic I/O

- Streams
- User I/O

## 3. Java Collections

- Introduction
- Data structures
- Java Collections Framework

# Exceptions

## Exception definition (I)

### Errors happen

- Code execution generates errors
- We must expect errors to happen
- We need a mechanism to handle errors

**Exception:** An error that disrupts the normal execution flow

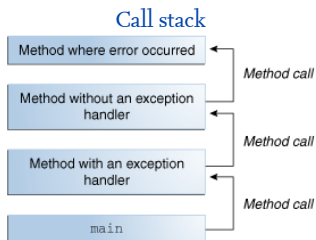
- File not found, division by zero, invalid argument, etc
- Code cannot be executed

Exceptions are an elegant solution to handle errors

- They are objects

# Exceptions

## Exception definition (II)

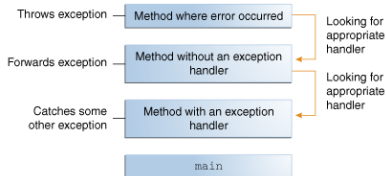


Call stack: Sequence of invoked methods

# Exceptions

## Exception definition (III)

### Exception handling



When an error happens ...

1. An exception is thrown
2. Code execution is stopped
3. The JVM goes back in the call stack
4. When the JVM finds an exception handler, it is executed

The exception handler catches the exception, the program finishes otherwise

# Exceptions

## Exception definition (IV)

```
Exception in thread "main" java.lang.IllegalArgumentException:  
    at org.ph.training.Class2.call(Class2.java:12)  
    at org.ph.training.Class1.call(Class1.java:14)  
    at org.ph.training.JavaSTSimulator.main(JavaSTSimulator.  
        java:20)
```

# Exceptions

## try-catch (I)

Handling an exception requires a try-catch statement

- try: Encloses the vulnerable code
- catch: Code that handles the exception

### try-catch statement

```
try {  
    // Risky code  
} catch (ExceptionType name) {  
    // Handle error  
} catch (ExceptionType name) {  
    // Handle error  
}
```



# Exceptions

## try-catch (II)

### ListOfNumbers.java (compilation error!)

```
public class ListOfNumbers {  
    private List<Integer> list;  
    private static final int SIZE = 10;  
  
    public ListOfNumbers () {  
        list = new ArrayList<Integer>(SIZE);  
        for (int i = 0; i < SIZE; i++)  
            list.add(new Integer(i));  
    }  
  
    public void writeList() {  
        PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));  
        for (int i = 0; i < SIZE; i++)  
            out.println("Value at:" + i + "=" + list.get(i));  
        out.close();  
    }  
}
```

# Exceptions

## try-catch (III)

### ListOfNumbers.java (corrected)

```
try {
    System.out.println("Entered try statement");
    out = new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < SIZE; i++) {
        out.println("Value at: " + i + "=" + list.get(i));
    }

} catch (FileNotFoundException e) {
    System.err.println("FileNotFoundException: "
        + e.getMessage());
    throw new SampleException(e);

} catch (IOException e) {
    System.err.println("Caught IOException: "
        + e.getMessage());
}
```

# Exceptions

## try-catch (IV)

### finally statement example

```
public void writeList() {
    PrintWriter out = null;
    try {
        System.out.println("Entering");
        out = new PrintWriter(new FileWriter("Out.txt"));
        for (int i = 0; i < SIZE; i++)
            out.println("At: "+i+"="+vector.elementAt(i));
    } catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("Invalid index: " + e.getMessage());
    } catch (IOException e) {
        System.err.println("IO error:" + e.getMessage());
    } finally {
        if (out != null) {
            System.out.println("Closing PrintWriter");
            out.close();
        } else {
            System.out.println("PrintWriter not open");
        }
    }
}
```

# Exceptions

## Exceptions thrown by a method (I)

Sometimes, we do not know how to handle an exception

- It is better to raise the exception
- Good practice: Handle exceptions when you know what to do

Methods can throw exceptions

- Forces handling errors
- Forces good programming

### Method throwing an exception

```
public void method() throws Exception {  
    // Code  
}
```

# Exceptions

## Exceptions thrown by a method (II)

### Example

```
public void writeList() throws IOException {  
    PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));  
  
    for (int i = 0; i < SIZE; i++) {  
        out.println("Value at: " + i + " = " + vector.elementAt(i));  
    }  
  
    out.close();  
}
```

# Exceptions

## Exceptions thrown by a method (III)

### Exception throwing

- Automatic: Certain operations like dividing by zero
- Manual: Using **throw** statement

Remember: Exceptions are objects

### Example

```
public Object pop() {  
    Object obj;  
  
    if (size == 0) throw new EmptyStackException();  
  
    obj = objectAt(size - 1);  
    setObjectAt(size - 1, null);  
    size--;  
    return obj;  
}
```

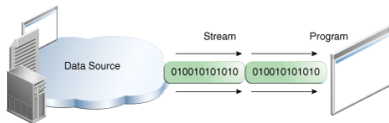
# Basic I/O

## Streams (I)

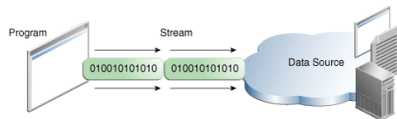
All I/O operations in Java are based on **streams**

- Stream: A sequence of data
- Input and output streams

### Input stream



### Output stream

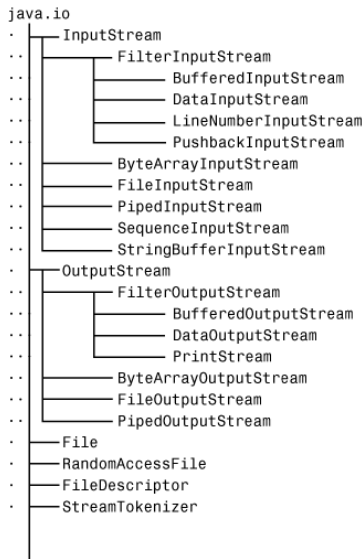


Data may come from or go to anywhere

- File, device, network, ...

# Basic I/O

## Streams (II)



Java I/O has a complex class hierarchy



# Basic I/O

## User I/O (I)

By default, JVM has three streams:

- Input stream (`System.in`): Class `InputStream`
- Output stream (`System.out`): Class `PrintStream`
- Error stream (`System.err`): Class `PrintStream`

Problem: `InputStream` reads bytes, but not characters or strings

- The solution is to transform it into a `BufferedReader` object

# Basic I/O

## User I/O (II)

### IO Example

```
public static void main(String args[]) {  
    InputStreamReader isr = new InputStreamReader(System.in);  
    BufferedReader br = new BufferedReader(isr);  
  
    while (true) {  
        double number;  
  
        try {  
            System.out.print("Number: ");  
            String str = br.readLine();  
            number = Double.valueOf(str).doubleValue();  
        } catch (NumberFormatException nfe) {  
            System.out.println("Not a number!");  
            continue;  
        } catch (IOException e) {  
            System.out.println("IO error" + e.getMessage());  
            continue;  
        }  
  
        System.out.println("Number: " + number);  
        return;  
    }  
}
```

# Java Collections

## Introduction

Programming is about information representation

- Simple data are easy to represent:
  - Numbers, characters, strings, etc

Reality uses to be more complicated: Classes

- How can we store several objects?
- How can we represent complex data?

We need more powerful mechanisms to store information: Data structures

# Java Collections

## Data structures: Array

Vector (1-D array)

0	1	2	3
$a_0$	$a_1$	$a_2$	$a_3$

Matrix

	0	1	2	3
0	$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
1	$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
2	$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$

### Advantajes:

- Very fast
- No extra memory
- Native language support

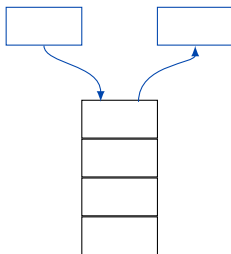
### Disadvantajes:

- Fixed size

# Java Collections

## Data structures: Stack and queue

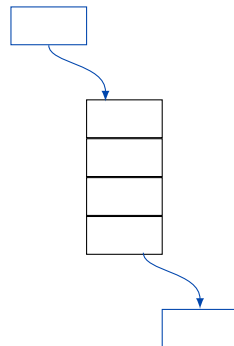
Stack



### Operations

- `pop()` and `push()`

Queue



### Operations

- `enqueue()` and `dequeue()`

# Java Collections

## Data structures: Lists

Linked list



Hash table

Key <sub>1</sub>	Value <sub>1</sub>
Key <sub>2</sub>	Value <sub>2</sub>
Key <sub>3</sub>	Value <sub>3</sub>

### Operations

- `put()` and `get()`
- `remove()`

(List in Python)

### Operations

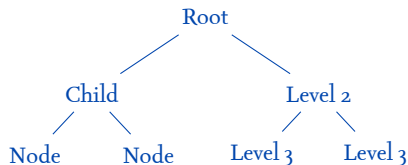
- `put()` and `get()`
- `remove()`

(Dictionary in Python)

# Java Collections

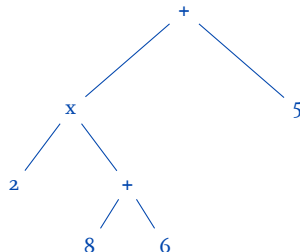
## Data structures: Trees (I)

### Trees



### Operations

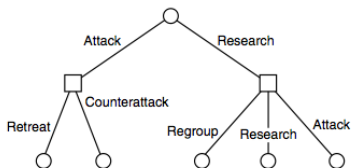
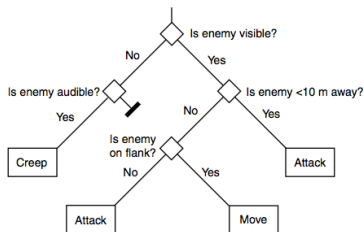
- `insert()` and `remove()`
- `search()`



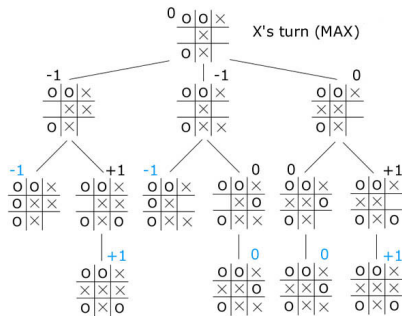
$$2 * (8 + 6) + 5$$

# Java Collections

## Data structures: Trees (II)



Source: Ian Millington, John Funge. "Artificial Intelligence for Games". Ed. Morgan-Kaufmann. 2009.



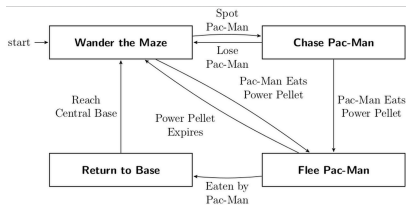
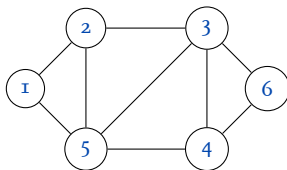
(Source)



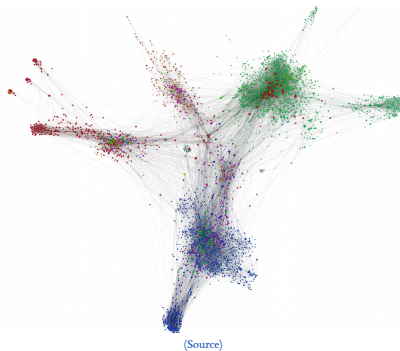
# Java Collections

## Data structures: Graphs

### Graphs



(Source)



(Source)

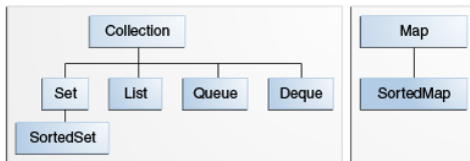
# Java Collections

## Java Collections Framework

- Java Collections is a framework that implements data structures
  - It is very useful
  - Equivalent in Java to C++'s STL
- A collection is a group of objects ... regardless of their class
- Three components
  1. Interfaces: Exposes the collection interface
  2. Implementations: The implementation of a interface
  3. Algorithms: Useful operations like sorting

# Java Collections

## Java Collections Framework: Interfaces



(Source)

- **Collection:** The root of the hierarchy
- **Set:** A collection without duplicates, not sorted
  - **SortedSet:** A set with sorted elements
- **List:** Ordered with duplicates and positions
- **Queue:** Insertion and extraction only
- **Map:** Key-value data structure, no duplicates, not sorted
  - **SortedMap:** A map with order

# Java Collections

## Java Collections Framework: Collection interface

### Methods in the Collection interface

- `int size();`
- `boolean isEmpty();`
- `boolean contains(Object element);`
- `boolean add(E element);`
- `boolean remove(Object element);`
- `Object[] toArray();`

### Collection iteration

```
for (Object o : collection)
    System.out.println(o);
```

# Java Collections

## Java Collections Framework: Set interface

- Same methods than `Collection`, no duplicates
- Implementations:
  - `HashSet`, `TreeSet`, `LinkedHashSet`

### FindDups

```
public class FindDups {
    public static void main(String[] args) {
        Set<String> s = new HashSet<String>();

        for (String a : args)
            if (!s.add(a))
                System.out.println("Duplicate detected");

        System.out.println(s.size() + " distinct words: "
            + s);
    }
}
```

# Java Collections

## Java Collections Framework: List interface

Same methods than `Collection`, no duplicates, ordered

- `E get(int index);`
- `E set(int index, E element);`
- `int indexOf(Object o);`
- `int lastIndexOf(Object o);`

Implementations: `ArrayList` and `LinkedList`

### List example

```
List<String> list = new ArrayList<String>(c);  
list.add("hola");  
System.out.println(list.size());
```

# Java Collections

## Java Collections Framework: Map interface (I)

Stores key-value pairs, same methods than `Collection`, and new ones:

- `V put(K key, V value);`
- `V get(Object key);`
- `V remove(Object key);`
- `boolean containsKey(Object key);`
- `boolean containsValue(Object value);`

Implementations: `HashMap`, `TreeMap` and `LinkedHashMap`

# Java Collections

## Java Collections Framework: Map interface (II)

### Freq.java

```
public class Freq {
    public static void main(String[] args) {
        Map<String,Integer> m =
            new HashMap<String,Integer>();

        for (String a : args) {
            Integer freq = m.get(a);
            m.put(a, (freq == null) ? 1 : freq + 1);
        }

        System.out.println(m.size() + " distinct words:");
        System.out.println(m);
    }
}
```