# Advanced Programming in Java
# CM1: the Language

Arthur Bit-Monnot

INSA 4IR

Section 1

# Structure and Objectives of the Course

# Course Objectives

Build skills to scale up the development process.

Objectives:

- provide tools for **software development at medium scale** (small team of contributors)
- boost **general programming skills**, based on the Java language
- understand the **life cycle of a Java program in the JVM** (JIT, garbage collector, . . . )

# Course Objectives

Build skills to scale up the development process.

Objectives:

- provide tools for **software development at medium scale** (small team of contributors)
- boost **general programming skills**, based on the Java language
- understand the **life cycle of a Java program in the JVM** (JIT, garbage collector, . . . )

Taught through Java but mostly transferrable to other languages

# Topics Covered

- Build configuration
- tests and automatization
- dependency management
- error handling / thread safety
- reproducibility
- code structure (package, public/private)
- sources management
- release and versioning (interface stability)

# Context: Training Unit

| UE | Responsable | Code | CM | TD | TP |
|---|---|---|---|---|---|
| **Conception et programmation avancées** | **S. Yangui** | **I4IRIL11** | **18,75** | **43,75** | **27,5** |
| UML et patrons de conception | N. Van Wambeke | | 10 | 16,25 | |
| Programmation avancée en Java | A. Bit-Monnot | | 7,5 | 13,75 | 27,5 |
| Conduite de projet | N.Guermouche | | | 5 | |
| Processus de développement logiciel automatisé | N. Guermouche | | 1,25 | 8,75 | |

2 phases:

1. independent training for each sub-area (java, UML, . . . )
2. A common project: the ChatSystem

# Planning: Advanced Programming, Phase 1

**Courses** (3 CM)

- The Java Language
- The Java Virtual Machine
- Development process

**Labs** (6 TD)

- Threads / Concurrency
- Networking (TCP / UDP)
- Graphical User Interface (GUI)

# Planning: Advanced Programming, Phase 2

**Application Project: Decentralized ChatSystem**

- Contact discovery
- Message exchange
- History management
- Graphical User Interface

# Planning: Advanced Programming, Phase 2.1

- Focus on **Contact Discovery**

Involves:

1. Conception of the module (3 TD UML)
2. Implementation of the module (4 TP Prog)
3. Project Management (2 TD)

=> Code Milestone

# Planning: Advanced Programming, Phase 2.2

- Building on your "Contact discovery" struggles

Present solutions to your problems in contact discovery:

- build configuration
- tests
- logging
- design patterns
- thread safety
- ...

Format:

- Code Repository + Pre-recorded live coding (2CM, autonomous)
- Feedback / Evaluation formative (1 TD)

# Planning: Advanced Programming, Phase 2.3

- Full application

Involves:

1. Complete Conception (4 TD UML)
2. Implementation (6 TP Prog)
3. Project Management (2 TD)

=> Final Report + Code

Section 2

# Java in the Realm of Programming Languages

# Object-Oriented Programming

Programs are composed of **datatypes** and **code**

- in OOP, datatypes and code are grouped into the same construct: a class

The *instance of a class* is called an **object**

- an object has fields, defined in the class (`count`)
- an object has *methods*, defined in the class (`increase`, `get`)

Methods have privileged access to the object's fields.

```
public class Counter {
    private int count = 0;

    public void increase() {
        this.count++;
    }

    public int get() {
        return this.count;
    }
}
```

# Quizz

```java
public class Counter {
    private int count = 0;

    public void increase() {
        this.counter++;
    }

    public int get() {
        return this.counter;
    }
}
```

What are the methods available for a `Counter` instance?

# Quizz

```
public class Counter {
    private int count = 0;

    public void increase() {
        this.counter++;
    }

    public int get() {
        return this.counter;
    }
}
```

What are the methods available for a `Counter` instance?

- `increase()`
- `get()`

# Quizz

```
public class Counter {
    private int count = 0;

    public void increase() {
        this.counter++;
    }

    public int get() {
        return this.counter;
    }
}
```

What are the methods available for a `Counter` instance?

- `increase()`
- `get()`
- `clone()`
- `equals()`
- `finalize()`
- ...

**Inherited** from the `java.lang.Object` top-level class.

# Inheritance the heart of OOP

All (decently recent) programming languages have mechanism to couple datatypes and code

- BUT **inheritance** is the crucial feature that sets OOP apart.

Key benefits:

- code sharing between related datatypes
- specialization of high-level behavior

Many drawbacks: OOP is not a global optimal in the programming language design space

# At the top was `java.lang.Object`

```
public class Counter {
```

Is implicitly rewritten to

```
public class Counter extends java.lang.Object {
```

Define the common capabilities of **all** objects in a program.

> java.lang.Object documentation

# At the top was `java.lang.Object`

Printing:

- `toString()`

Identity:

- `equals()`
- `hashCode()`

Copy

- `clone()`

Synchronization (concurrency)

- `notify()` / `notifyAll()`
- `wait()`

Runtime introspection:

- `getClass()`

Lifecycle management

- `finalize()` (deprecated)

# Building well-behaved objects: String representation

```java
public class MyInt {
    public final int n;

    MyInt(int n) {
        this.n = n;
    }
```

```java
MyInt a = new MyInt(1);
MyInt b = new MyInt(2);


// without toString
System.out.println(a) // MyInt@8efb846
System.out.println(b) // MyInt@2a84aee
```

# Building well-behaved objects: String representation

```java
public class MyInt {
    public final int n;

    MyInt(int n) {
        this.n = n;
    }

   @Override
   public String toString() {
      return this.n.toString();
   }
```

```java
MyInt a = new MyInt(1);
MyInt b = new MyInt(2);


// without toString
System.out.println(a) // MyInt@8efb846
System.out.println(b) // MyInt@2a84aee

// with toString
System.out.println(a) // 1
System.out.println(b) // 2
```

# Dynamic Dispatch for inheritance handling

```java
public static void print(Object o) {
    System.out.println("object: " + o.toString());
}

print(new Object()); // object: Object@4efe845
print(new MyInt(3)); // object: 3
print("hello");      // object: hello
```

# Dynamic Dispatch for inheritance handling

```
public static void print(Object o) {
    System.out.println("object: " + o.toString());
}

print(new Object()); // object: Object@4efe845
print(new MyInt(3)); // object: 3
print("hello");      // object: hello
```

The `print` function works for any object.

How can it select the right `toString()` method to call?

# Runtime Reflection

```
var a = new MyInt(1);
var clazz = a.getClass();
System.out.println(clazz);

System.out.println("Fields:");
for (var field : clazz.getDeclaredFields()) {
    System.out.println("  " + field);
}

System.out.println("Methods:");
for (var method : clazz.getDeclaredMethods()) {
    System.out.println("  " + method);
}

System.out.println("Super Class: "
    + clazz.getGenericSuperclass());
```

```
class test$MyInt

Fields:
  private final int test$MyInt.n

Methods:
  public int test$MyInt.toString()

Super Class: class java.lang.Object
```

# Runtime Reflection for Meta-Programming

Any java object provide access, **at runtime**, to its `Class` object and all metadata associated

- name
- fields
- methods
- superclasses
- ...

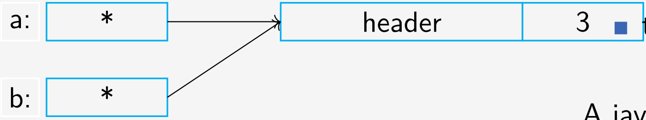
This is called **runtime reflection**

Example usage: > JSON encoder for any java object (in 30 lines)

```
toJson(new Point(13, 18));
// { x: 13, y: 18 }
```

# Anatomy of a Java Object: Enabler for Runtime Reflection

```java
MyInt a = new MyInt(3);
MyInt b = a;
```




Each java object has a memory space allocated **on the heap**.
Contains:

- a header (12/16 bytes)
- the object's fields values

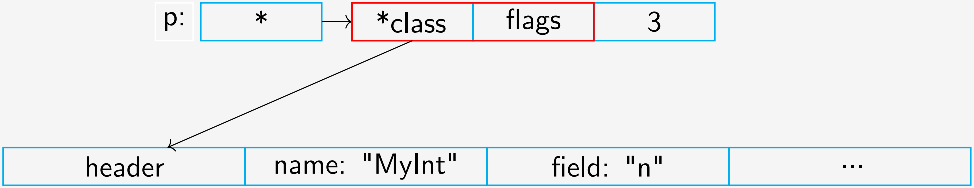
A java program manipulates **references** (~ pointers) to the object's memory space.

So what's in the header?

# The Java Object Header

The java object header is composed of two parts[1]

- a pointer to the **class object**
- 4/8 bytes of flags used internally by the JVM (synchronization, garbage collection)




> `java.lang.Class` documentation

[1] may depend on the implementation of the JVM

# Virtual Table for Dynamic Dispatch

In the class object, there is a **virtual table** that associates each method available on the class to its implementation:

Example virtual table for our `MyInt` object

| Method | Implementation |
|---|---|
| equals | Object::equals (@21ab324d) |
| hashCode | Object::hashCode (@47b3f24d) |
| toString | MyInt::toString (@63c3ef9) |

# Dynamic Dispatch, putting it all together

```
public static void print(Object o) {
    System.out.println("object: " + o.toString());
}
```

To invoke `o.toString()`, the JVM must:[2]

- dereference the pointer `o`, to access the object's memory space in the heap
- dereference the *class pointer* in the object's header, to access the class' metadata
- find the `toString` method in the *virtual table*
- **execute the code** at the address pointed at in the virtual table

Called **dynamic dispatch** as it allows to dynamically decide which function to call based on the runtime class of an object.

[2] In the worst case, many optimizations may avoid part of these costs

# Dynamic Dispatch, caveats

Dynamic dispatch is a **corner stone of OOP**

$\Rightarrow$ required any time a method can be *overridden*

# Dynamic Dispatch, caveats

Dynamic dispatch is a **corner stone of OOP**

⇒ required any time a method can be *overridden*

Caveats:

- finding the method to call is costly (two extra indirections)
  - can be greatly mitigated by caching
- prevents compiler optimization (inlining)
  - partially mitigated by specific compiler techniques (devirtualization)
- Impose an inefficient memory layout

# Memory Overhead of Java Objects

```java
MyInt a = new MyInt(7);
```

a represents the number 7, a 4 bytes integer.

What is the memory overhead of having be a java object?

[3]Conservative estimate, may be 8 bytes on 64bits systems without compressed oops

# Memory Overhead of Java Objects

```
MyInt a = new MyInt(7);
```

a represents the number 7, a 4 bytes integer.

What is the memory overhead of having be a java object?


- pointer to object memory space: 4 bytes[3]
- header:
  - flags: 8 bytes:
  - class pointer: 4 bytes

[3] Conservative estimate, may be 8 bytes on 64bits systems without compressed oops

# Memory Overhead of Java Objects

In the best case, there is an overhead of 16 bytes per object.

- With `MyInt` that is 80% of memory overhead

**Worse,** the payload is hidden behind a pointer !!!!

- requires additional memory access, and potential cache-miss[4]

[4] Recall that an uncached memory read is 100 times slower than an arithmetic operation

# Memory Overhead of Java Objects

In the best case, there is an overhead of 16 bytes per object.

- With `MyInt` that is 80% of memory overhead

**Worse,** the payload is hidden behind a pointer !!!!

- requires additional memory access, and potential cache-miss[4]

The penalty is **dramatic** for basic types.

- cannot be escaped if you want to maintain *runtime polymorphism* ($\sim$ dynamic dispatch)

[4] Recall that an uncached memory read is 100 times slower than an arithmetic operation

# “Everything is an `Object`”... except primitive types

Java deeply adheres to the “*everything is an object*” moto...

- but provides an escape hatch for performance critical code

**Primitives types**:

- int / long
- float / double
- bool
- char

Principle:

- a primitive type only holds a value

```java
int i = 4;
```


```java
long l = 4;
```

# Boxing: primitive $\leftrightarrow$ object interoperability

```
public static void print(Object o) {
    System.out.println("object: " + o);
}

int n = 7;
print(n); // problem: print only accepts object references
```

# Boxing: primitive $\leftrightarrow$ object interoperability

```
public static void print(Object o) {
    System.out.println("object: " + o);
}
```

```
int n = 7;
print(n); // problem: print only accepts object references
```

```
// the above is implicitly translated as
int n = 7;
Integer nBoxed = new Integer(n);
print(nBoxed);
```

Called **boxing**: encapsulation of a pure value into an object

- enables (costly) interoperability of primitive types with normal java code

# Boxing: available for all primitives

| Primitive type | Boxed type |
|---|---|
| `int` | `java.lang.Integer` |
| `long` | `java.lang.Long` |
| `float` | `java.lang.Float` |
| `double` | `java.lang.Double` |
| `boolean` | `java.lang.Boolean` |
| `char` | `java.lang.Char` |

# Java Objects, a small recap

- a java program manipulates **references** to objects
- each object has its own **identity** and **memory space** on the heap
- the **class** of the object can be retrieved at runtime thanks to the **class pointer** in the **object header**
  - enables **runtime reflection**: analysis of the object's metadata
  - enables **dynamic dispatch**, choice at runtime of which method to execute based on the object's runtime class

# Building well-behaved objects: String representation

```java
public class MyInt {
    public final int n;

    MyInt(int n) {
        this.n = n;
    }

   @Override
   public String toString() {
      return this.n.toString();
   }
```

```java
MyInt a = new MyInt()
System.out.println(a) // 1
System.out.println(b) // 2
```

# Building well-behaved objects: Equality

Two kinds of equality tests in java:

- `a == b`: test whether two objects have the same **identity** (~ pointer equality)
- `a.equals(b)`: test whether two object are **logically equivalent**
  - provided with a dummy default in `java.lang.Object`
  - can (and should) be overridden

> Object::equals documentation

# Building well-behaved objects: Equality

The `equals` method should return true if two objects are equivalent and be:

- *reflexive*: `x.equals(x)`
- *symmetric*: `x.equals(y)` implies `y.equals(x)`
- *transitive*: `x.equals(y)` and `y.equals(z)` implies `x.equals(z)`
- *consistent*: stable result when changes to data
- return false if passed a null value
- **be consistent with `hashCode()`**

# Building well-behaved objects: `hashCode`

The *hash-code* is a 32 bits *signature* of an object which can be changed by overriding `Object::hashCode`

```
public int hashCode() { ... }
```

> `Object::hashCode` documentation

Properties:

- Two equivalent objects (as witnessed by `equals`) MUST have the same hashcode
- IDEALLY two distinct objects SHOULD have a different hashcode

# Building well-behaved objects: `hashCode`

Exercise: how to get (well distributed) hashes for strings ?

`"a", "b", "ab", "aab", "ba", ...`

# Building well-behaved objects: hashCode

Exercise: how to get (well distributed) hashes for strings ?

```
"a", "b", "ab", "aab", "ba", ...
```

```
int hash(String str) {
    int result = 1; // 1: meaningful hashCode even if we get an array of 0
    for (char c : str) {
        result = 31 * result + (int) c;  // 31: magic prime number
    }
    return result;
}
```

```
// built-in method for combining hashes of several objects
Objects.hash("Arthur", 35, "INSA Toulouse", 12.34)
```

# Building well-behaved objects: Equality & Hash

```
public class MyInt {
    public final int n;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        MyInt myInt = (MyInt) o;
        return n == myInt.n;
    }

    @Override
    public int hashCode() {
        return Objects.hash(n);
    }
```

Section 3

# The Standard Library (stdlib)

# What's a standard library?

**stdlib**: a collection of code that is **part of the language definition**

Purpose:

- *code sharing*: provide common code that most programs use
  - file handling, collections, . . .
- *interoperability*: ensure that programs agree on fundamental types

Because it is part of the language, the compiler knows the types of the stdlib

- enables syntactic sugar

# Syntactic Sugar for StdLib

```
var i = new MyInt(3);
System.out.println("num: " + i);
```

# Syntactic Sugar for StdLib

```java
var i = new MyInt(3);
System.out.println("num: " + i);
```

The compiler knows:

- that `"num: "` is a `java.lang.String`
- that `i` is an instance of `java.lang.Object`
- that `java.lang.Object` has the `toString()` method

and can interpret the above code as:

```java
System.out.println("num: " + i.toString());
```

# What's in the java standard library

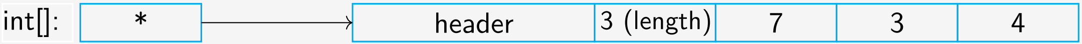
- language fundamentals: `Object`, `String`, `Integer`, ...
- concurrency: `Thread`, locks, atomics
- collections: lists, sets, maps
- input/output: files
- networking: TCP, UDP
- graphical user interface

# StdLib: Arrays

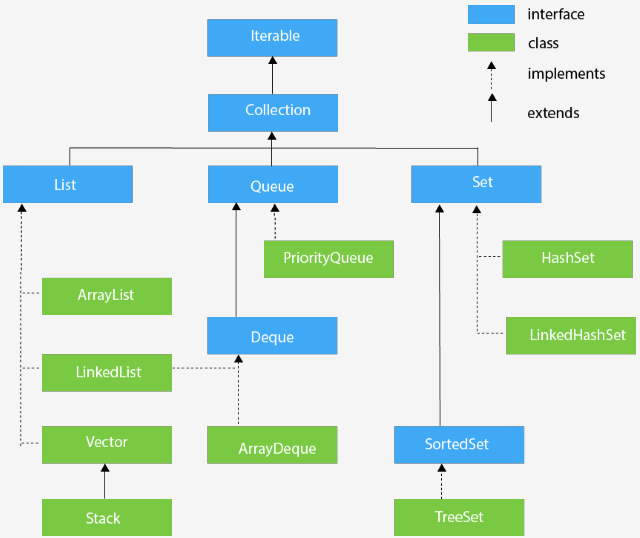
- An Array is a **fixed-length** block of memory
- Contains:
  - a header (like all java objects)
  - a field indicating the length of the array
  - one space for each element of the array

Exists for

- primitive types (`int[]`, `float[]`, `...`)
- Object types (`Object[]`, `MyInt[]`, `...`)



# StdLib: The Collection Library



# The Iterable Interface

> Iterable documentation

```
public void printAll(Iterable<?> iterable) {
    for (Object item : iterable) {
        System.out.println(item)
    }
}
```

# The Iterable Interface

> Iterable documentation

```
public void printAll(Iterable<?> iterable) {
    for (Object item : iterable) {
        System.out.println(item)
    }
}

// desugared
public void printAll(Iterable<?> iterable) {
    Iterator<?> iterator = iterable.iterator();
    while (iterator.hasNext()) {
        Object item = iterator.next();
        System.out.println(item)
    }
}
```

# Generics

Most collections are *generic*, they may contain any **reference type** (not primitive types)

In `List<T>`, `T` is a **type parameter**.

- It indicates what is the class of all objects contained in the list.

```
List<MyInt> ints = new ArrayList();
ints.add(new MyInt(7));
```

In a receiver position, you may indicate `?` as a type parameter:

- indicates that your program will work regardless of the type

```
public void printAll(Iterable<?> iterable) {
```

# Different collections

- list: sequence of values
- set: group of unordered, unique values
- queue: collection of values for which there is an extraction order
- map: dictionary associating keys to values

# Case Study: HashSet<Integer>

Set<T>: A collection that contains no duplicate elements.

- insert(T t)
- contains(T t)
- remove(T t)

HashSet<T>: a Set<T> backed by a hash table

```
class MyInt {
    public final int n;

    private MyInt(int n) {
        this.n = n;
    }

    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.g
        MyInt myInt = (MyInt) o;
        return n == myInt.n;
    }

    public int hashCode() {
        return n; // terrible implementati
    }
}
```