

# Introduction to Systems Programming

## Leaving BlueJ

*For this lecture, I suggest reading from the book "Program Development in Java".*

2025 - Simón Gutiérrez Brida *(Based on material by Dr. Nazareno Aguirre)*

# Java without BlueJ (nor any IDE)

Being dependant on IDEs to write/compile/and execute code is far from ideal. Even though IDEs represent a very useful tool for a developer, one must be able to write/compile/execute code without one.

Let's start by writing our first code without any IDE.

# The main method

Any Java Program will start from a method with this signature:

```
public static void main(String[] args)
```

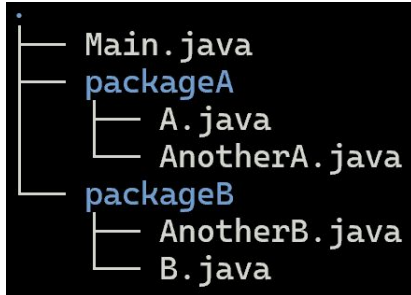
Let's analyze this signature:

- It is public, so anyone can access it.
- It has void as the return type, so it doesn't return anything.
- It has an array of String as an argument.
- It uses the **static** keyword, it means that with method is a "Class Method", is not applied/invoked on any object. When applied to a field, it will mean a "Class Field", every object will share the same instance of that field.
- The `args` argument will hold all command-line arguments as strings.

# Packages

Packages allows us to modularize our codebase into logical groups of classes. From the perspective of an OS, a package is just a path representing a folder with java files inside. From the perspective of Java, packages are a sequence of names divided by a dot (".").

Example:



```
package packageA;

public class A {

    public String greetings() {
        return "Hi, I'm A!";
    }

}
```

```
import packageA.A;
import packageA.AnotherA;
import packageB.B;
import packageB.AnotherB;

public class Main {

    public static void main(String[] args) {
        A a = new A();
        AnotherA anotherA = new AnotherA();
        B b = new B();
        AnotherB anotherB = new AnotherB();
        System.out.println(a.greetings());
        System.out.println(anotherA.greetings());
        System.out.println(b.greetings());
        System.out.println(anotherB.greetings());
    }

}
```

# Compilation and Execution

Even though Java is interpreted, it is interpreted by a Java Virtual Machine (JVM) which doesn't take Java code (.java files) as input, but Java bytecode (.class files).

A compilation process is therefore needed to compile Java source code, into Java Bytecode.

A compiled Java Code will not be executable as is, it is the JVM which executes it (interprets it).

For these two tasks we will use the Java Compiler (javac) and the java command to launch a JVM (java) with our class file (this class file must be the one that defined the “main” method we saw earlier).

# Demo

Let's practice with what  
we saw so far.

# Introduction to Systems Programming

## Postconditions and Class Invariants

*For this lecture, I suggest reading from the book "Program Development in Java".*

# Having full contracts

For the moment we have used documentation and preconditions to provide an abstraction of a method or class. With these two tools we are able to give a readable explanation of what a specific class is and what each method does without the client of the class needing to understand how everything is implemented.

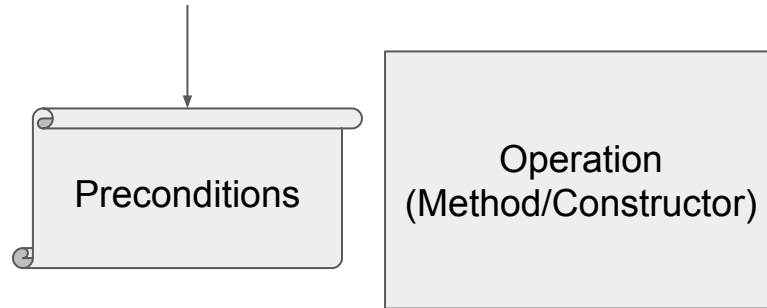
Preconditions acted as a contract that only saved the developer for bad usage of a method/constructor by the client of that class.

The client still don't have any "guarantees" for when the client satisfies the preconditions but the operation does not work.



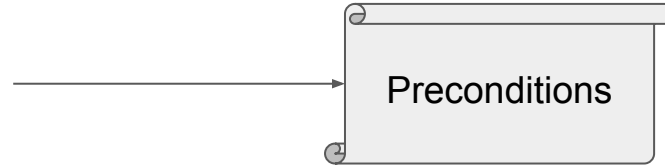
# Preconditions and Postconditions

The client must satisfy these.



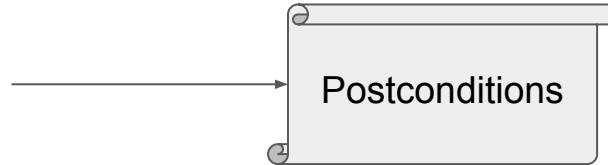
# Preconditions and Postconditions

The client must satisfy these.

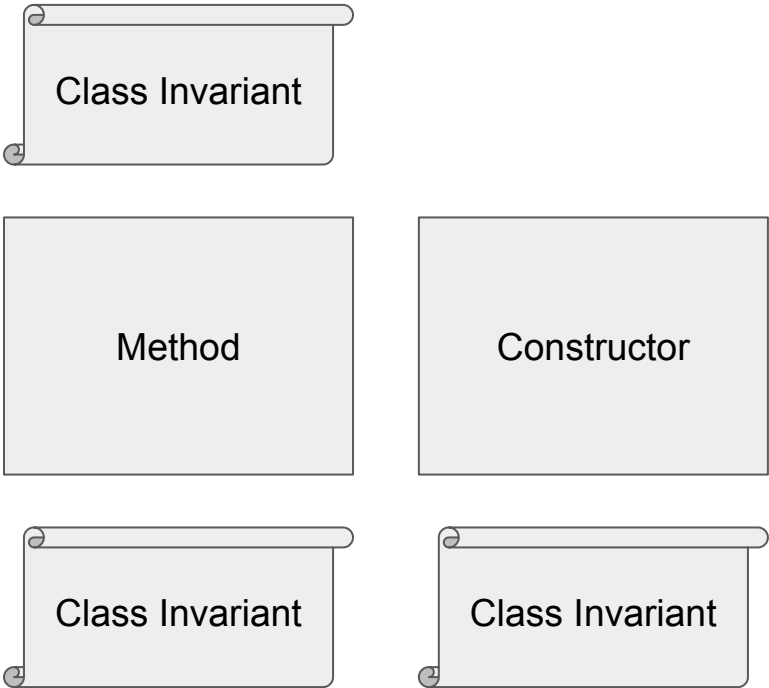


If preconditions are satisfied, the operation must satisfy the postconditions when finished.

The developer must guarantee that these will be satisfied if the preconditions are.



# Class Invariants



Class Invariant

Method

Constructor

Class Invariant

Class Invariant

Class invariants specify all correct states for an object of a particular class.

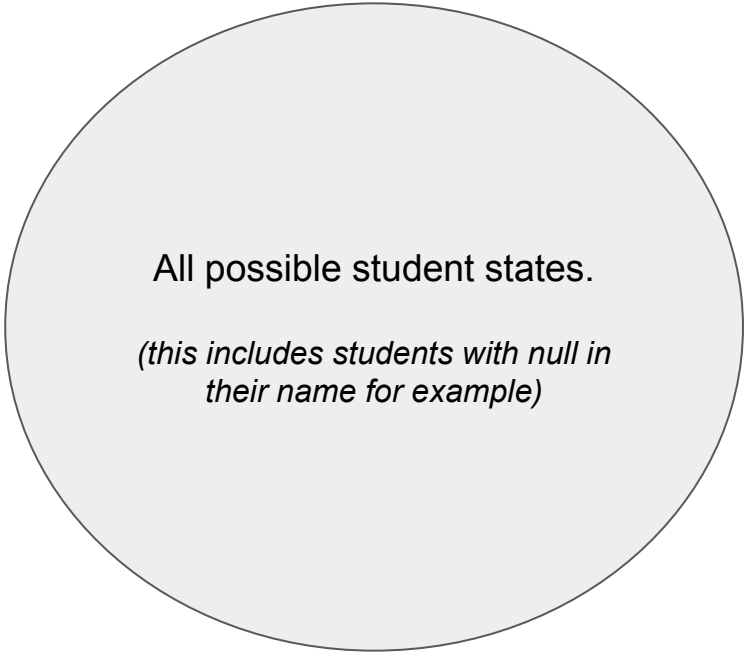
A class invariant filters what states are valid representations for objects of a particular class.

# Class Invariants, an example

```
public class Student {  
    private String name;  
    private String surname;  
    private int age;  
    private int id;
```

```
//TODO: constructors and methods
```

```
}
```



All possible student states.

*(this includes students with null in  
their name for example)*

# Class Invariants, an example

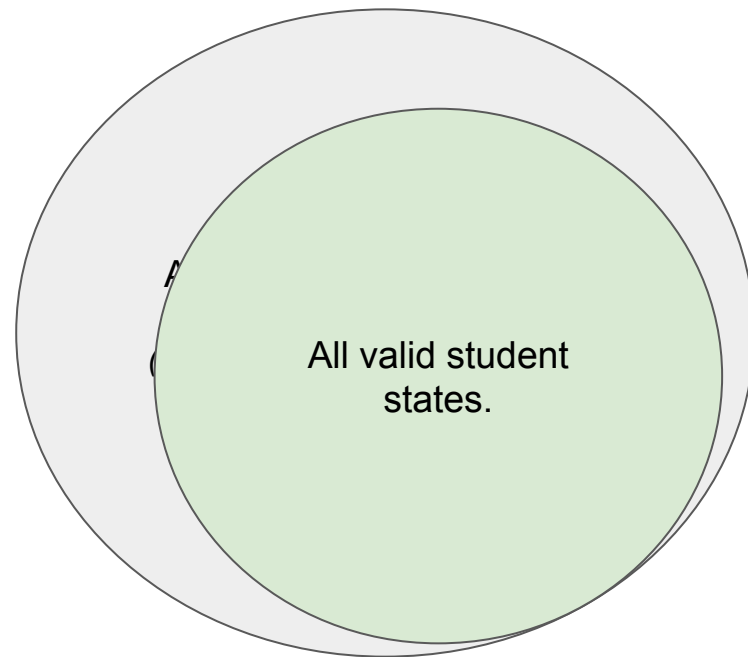
```
public class Student {  
    private String name;  
    private String surname;  
    private int age;  
    private int id;  
  
    //TODO: constructors and methods  
  
}
```

Class Invariant:

- Name and surname must not be null, contain only blanks, start or end with blanks, contain any character that is not a letter.
- Age must be strictly greater than zero (that will depend on the country).
- Id must be a valid id format, it will depend on the software requirements.

# Class Invariants, an example

```
public class Student {  
    private String name;  
    private String surname;  
    private int age;  
    private int id;  
  
    //TODO: constructors and methods  
}
```



# Checking Class Invariant

Is usual, and recommended, to return false as soon as possible.

```
public boolean repOk() {  
    //returns true iff all invariants are satisfied  
}
```

# Demo

Let's write an Invariant  
for Clock and  
BoundedCounter.



# Introduction to Systems Programming

## Intro to exceptions

*For this lecture, I suggest reading from the book "Program Development in Java".*

# Exceptions

Java uses exceptions to manage erroneous behaviour. Any runtime error in Java will throw an exception. There are checked and unchecked exceptions. Checked exceptions require to be declared in a methods/constructor profile, and require the developer to write code to catch them.

Even if one doesn't write code to throw an exception in Java, exceptions can still be thrown, e.g.: dividing by zero (`ArithmeticException`); using the dot operator on a null value (`NullPointerException`); using an unsupported operation (`UnsupportedOperationException`), this exception is often used as the default code for any operation than needs to be implemented; invalid arguments for an operation (`IllegalArgumentException`); invalid object state for an operation (`IllegalStateException`).

*One can create a new exception, although we will use existing ones.*

# Exceptions - Example

```
public void removeAt(int index) {  
    if (index < 0 || index >= size) {  
        throw new IllegalArgumentException("index must be between 0 and " + (size - 1) + " but is " + index + " instead");  
    }  
    //CODE  
    if (!repOk()) {  
        throw new IllegalStateException("removeAt(" + index + ") broke class invariant");  
    }  
}
```

# Catching, rethrowing exceptions

Let's consider a custom, checked, exception called `CheatingIsBadMkay`. We have a checked exception (`IllegalAccessException`) which we want to rethrow as a `CheatingIsBadMkay` exception.

```
private static void copyFieldValue(Object original, Object target, Field field) throws CheatingIsBadMkay {
    boolean oldAccessibleStatus = setAccessibleStatus(field, true);
    try {
        field.set(target, field.get(original));
        setAccessibleStatus(field, oldAccessibleStatus);
    } catch (IllegalAccessException e) {
        throw new CheatingIsBadMkay("An error occurred while trying to access " + field.getName() + " field");
    }
}
```

Using try-catch, we can “try” to execute some code, and if an exception is thrown somewhere in that code, we can “catch” it and do something, here we are rethrowing the exception by encapsulating it into another.

# Catching, with unchecked exceptions

A less convoluted example, let's consider we want to transform a string representation of a number into an actual number, but return a default value if the transformation fails.

```
public int fromString(String string, int defaultValue) {  
    if (string == null) {  
        throw new IllegalArgumentException("string argument cannot be null");  
    }  
    try {  
        return Integer.parseInt(string);  
    } catch (NumberFormatException nfe) {  
        return defaultValue;  
    }  
}
```

# Introduction to Systems Programming

## Equality vs Identity

# Equality in primitive types

On primitive types, using the equality operator, `==`, means comparing two values. Example: `2 == 3` (false), `42 == 42` (true), `'a' != 'b'` (true).

On non-primitive types, using the equality operator, `==`, means comparing if two references are the same. Is like comparing pointers in C. In Java, each object has an unique ID called a hashcode, any two objects will return true when using the `==` operator iff their references are the same.

```
A a = new A();  
A b = a;  
System.out.println("is a == b true? " + a == b);
```

# Equality in primitive types

On non-primitive types, using the equality operator, `==`, means comparing if two references are the same. Is like comparing pointers in C. In Java, each object has an unique ID called a hashcode, any two objects will return true when using the `==` operator iff their references are the same.

```
A a = new A();  
A b = a;  
System.out.println("is a == b true? " + a == b);
```

This is usually not what we want. In general we consider two objects to be equal, if their states are equal.



# Equal method

Any class in Java will have some methods already implemented, although with a simple implementation, even if not written in the class' code.

We have already seen one of these method, **public** String **toString()**, this method is used to get a String representation of an object.

Another method is **public boolean equals**(Object other), this method takes an object and compares the current object to the argument. It takes an Object because there is no way to make this method in Java in any other way. Using Object makes the argument type compatible with any possible class.

To implement we will use this:

If an object **o** is one of the objects characterized by a class **C**, we say that:

- **o** is an **instance** of **C**
- **C** is the **(generating) class** of **o**

# Demo

Let's write an equals  
method (\*).

*(\*) Java has a built in operation called instanceof used as: `o instanceof Class` that returns true iff `o` is an instance of `Class`.*