Java Introduction

Prof. dr. Kris Luyten

Dries Cardinaels

Gilles Eerlings

Agenda

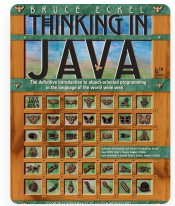
1. Java
2. Java
3. Java
4. Reading about Java
5. Programming in Java

Object-Oriented Programming

+

Basic Principles in Java

Required Reading Material



Thinking in Java (3rd ed.!)

Bruce Eckel

Pearson

- Chapter 1: Introduction to Objects; p. 36 - 61 (**understand & master**)
- Chapter 2: Everything is an Object; p. 85 - 114 (**only understand**)
- Chapter 3: Controlling Program Flow; review and consult when needed
- Chapter 4: Initialisation & Cleanup; "Guaranteed initialization with the constructor" and "Method overloading" (**understand & master**)

Basic Principles in Java

A Java program is a collection of (instantiated) classes

Classes and Objects

What exactly are these?

The Origins of the Object

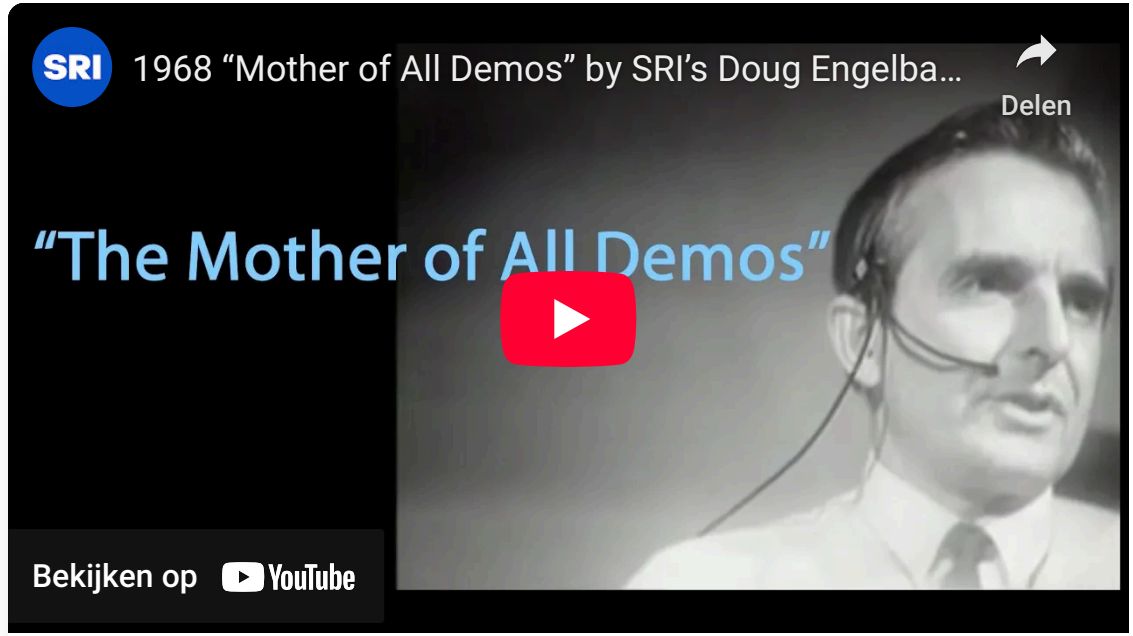
The original idea comes from **Sketchpad** (1960-1961) by *Ivan Sutherland*

[Assignment_02_01] Watch Sketchpad demonstration

The Mother of All Demos - Douglas Engelbart (1968)

At Stanford Research Institute, Engelbart revealed the oN-Line System (NLS), a collaborative computing vision that let teams compose, navigate, and share knowledge together. NLS demonstrated for the first time many fundamental elements of modern personal computing, including windows, hypertext, graphics, efficient navigation and command input, video conferencing, the computer mouse, word processing, dynamic file linking, revision control, and a collaborative real-time editor.

Source: SRI International — Augmentation Research Center archives & Wikipedia.



The Origins of the Object: Sketchpad

The original idea comes from

Sketchpad (1960-1961) by *Ivan Sutherland*

"Records with related procedures"

The Origins of the Object: Sketchpad

The original idea comes from

Sketchpad (1960-1961) by *Ivan Sutherland*

What else came from Sketchpad: GUIs, pen interaction, vector graphics, constraint-based geometry...

The Origins of the Object: Simula 67

The very first programming language with classes and objects

```
begin
  OutText("Hello, Simula world!");
  OutImage;
end;
```

The Origins of the Object: Simula 67

Simple class and object creation

```
class Greeter;
begin
  text msg;
  procedure init(t); text t;
  begin msg := t; end;

  procedure speak;
  begin
    OutText(msg);
    OutImage;
  end;
end;

begin
  Greeter g;
  g :- new Greeter;
  g.init("Hello from Simula!");
  g.speak;
end;
```

The Origins of the Object: Smalltalk

The first dynamic OO language

- Classes and objects become **"first class citizens"** of the programming language.
- **Object = Data + Operations** (+identity)

```
Object subclass: Hello [
    Hello >> greet [
        ^ 'Hello, World!'
    ]
]

| greeting message |
greeting := Hello new.
message := greeting greet.
message displayNl.
```

The state of the object + All possibilities to read or manipulate the state

The Origins of the Object

Smalltalk

Message = Receiver + Requested Operation

Who can receive the message...

+

...and what is being requested.

Messages vs Methods

Messages

- Dynamically bound to target (run-time)
- Are **sent**
- Example in Objective-C:

```
[calc calculate:x]
```

Methods

- Statically bound to target (compile time)
- Are **called**
- Example in Java:

```
calc.calculate(x)
```

equivalent

Messages vs Methods (Example)

Objective-C message send

The runtime can decide which implementation should run, or even forward the message to another object.

```
id calculator;

if (arc4random_uniform(2) == 1) {
    calculator = [ScientificCalculator new];
} else {
    calculator = [BasicCalculator new];
}

if ([calculator respondsToSelector:@selector(calculate:)]) {
    [calculator calculate:@42];
}
```

- Message dispatch resolves *at runtime*.
- @selector checks whether this object has a method called calculate:

Java method call

The compiler knows the class type and method signature, so the bytecode already contains the exact invocation to use.

```
Calculator calc = new ScientificCalculator();
calc.calculate(42); // resolved when the class is verified
```

- Method binding is determined *ahead of time* (static checking at compile time).
- Method calls use the type information available at compile-time - hard-wired in the compiled code.

Java

It should be **"simple, object-oriented and familiar"** It should be **"robust and secure"** It should be **"architecture-neutral and portable"** It should execute with **"high performance"** It should be **"interpreted, threaded, and dynamic"**

Early Java white paper

Java

- **Object-Oriented**
- **Class-based**
- **Methods**
- **General Purpose**
- **Garbage Collection**
- **Virtual Machine**
- **Platform Independent**
- **Syntax based on C/C++**

Java

- **No multiple inheritance**
- **No operator overloading**

Libraries, libraries, libraries

Collection of predefined packages available by default

- For data structures: **java.util** (Javadoc)
- For GUIs: **javax.swing** (Javadoc)
- For date/time formatting: **java.time.format** (`DateTimeFormatter`, `DateTimeFormatterBuilder`)
- For networking: **java.net** (Javadoc)
- Math, Collections, XML, SQL, Graphics, Security, I/O…

Code Example

```
import java.util.*;

public class Stuff {
    public static void main(String args[]) {
        Vector v = new Vector();
        v.addElement("blaai");
        v.addElement("vlaai");
        v.addElement("kers");
        System.out.println("size == " + v.size());
        System.out.println(v.elementAt(2));
    }
}
```

API Documentation

addElement

```
public void addElement(Object obj)
```

Adds the specified component to the end of this vector, increasing its size by one. The capacity of this vector is increased if its size becomes greater than its capacity.

This method is identical in functionality to the add(Object) method (which is part of the List interface).

Parameters:

- obj - the component to be added.

See Also:

- add(Object), List

Questions?

Deep Dive

Deep Dive

Connect Four in Java...

VierOpEenRij Class

```
package vieropeenrij;

/**
 * This class starts and manages the game logic. It uses the {@VorMatrix} as a model for the
 * game board and {@VierOpEenRijVenster} as the user interface for the model.
 * @author Kris Luyten
 */
public class VierOpEenRij {
    private VorMatrix $vorMatrix;
    public static int ROWS = 6;
    public static int COLS = 7;
    public static enum FILL { RED, YELLOW, EMPTY };
    private FILL $turn;

    /**
     * The main method – the application starts here
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        try {
            VierOpEenRij.getInstance().start();
        } catch(Exception e) {
            //code to catch all unexpected exceptions and handle them gracefully
        }
    }
```

VierOpEenRij Class (continued)

```
    /**
     * Starts the main thread of this application. Standard Java stuff.
     */
    public void start() {
        java.awt.EventQueue.invokeLater(new RunnableImpl());
    }

    private class RunnableImpl implements Runnable {
        public RunnableImpl() {
        }

        public void run() {
            $vorMatrix = new VorMatrix();
        }
    }
}
```

VorMatrix Class

```
package vieropeenrij;

/**
 * VorMatrix is the model of the game and contains the current state of the game board
 * @author Kris Luyten
 */
public class VorMatrix extends java.util.Observable {
    private VierOpEenRij.FILL $vorMatrix[][] =
        new VierOpEenRij.FILL[VierOpEenRij.COLS][VierOpEenRij.ROWS];

    /**
     * Creates an empty model for the board
     */
    public VorMatrix() {
        initMatrix();
    }

    /**
     * Initializes game board with empty buckets
     */
    private void initMatrix() {
        for(int i=0; i<VierOpEenRij.COLS; i++)
            for(int j=0; j<VierOpEenRij.ROWS; j++) {
                $vorMatrix[i][j] = VierOpEenRij.FILL.EMPTY;
            }
```

Questions?

Some Important ~~Tips~~ ~~Guidelines~~ Rules

1. Inheritance is perhaps the most difficult OO technique to apply correctly

- If you have doubts: prefer **associations over inheritance**
- Inheritance **reduces the flexibility** of your code and creates a **strong dependency** between classes ("high coupling" instead of "low coupling")

2. Methods are short

not more than 15 lines of code

(there are exceptions, but they are very rare)

3. Mainly Inspection and Mutation Methods

Make methods clearly recognizable and one of these two types:

- **Inspector**: returns a result that requires the object's data and changes nothing about the object's state. *"get" methods or getters*
- **Mutator**: changes the object's state (modifies a member variable, for example) and returns no result. *"set" methods or setters*

3. Mainly Inspection and Mutation Methods

- **Simple Division**
- **Code becomes much more readable and clearer**
- **Fewer bugs**

Methods that both modify an object’s state and return a value are a source of errors

3. Mainly Inspection and Mutation Methods

Language Support

C# Accessors

```
public string Name
{
    get
    {
        return name;
    }
    set
    {
        name = value;
    }
}
```

C++ const keyword

```
class TemperatureSensor {
public:
    void setThreshold(double value) { threshold_ = value; }
    double threshold() const { return threshold_; }

private:
    double threshold_{0.0};
};
```

Questions?

Exercises

If you find the assignments not clear enough: **make your own choices too**

Exercises

- **[assignment_02_01]** TIJ3 Chapter 2: exercises 1 & 7
- **[assignment_02_02]** Write a class `Matrix` in C++ that can represent a 3×3 matrix. Provide the necessary methods to add and multiply two matrices. Carefully apply the three rules of the previous slides. Let’s start now! Do this one yourself to assess yourself - DO NOT FOOL YOURSELF.
- **[assignment_02_03]** TIJ3 Chapter 3: exercises 7 & 9

Exercises

- **[assignment_02_04]**
  1. Download Connect Four example code
  2. Study the code **carefully**
  3. Use your favorite Java programming environment and compile the code
  4. Run the application.
  5. Change code (use your imagination)

[assignment_02_05] – Core Concepts Refresh

- Explain in your own words how Java's notion of a class differs from an object instance, and how message sending in Smalltalk relates to method calls in Java.
- Draft your answer offline first; write it down before asking for feedback.
- Consult two different LLMs, paste your answer and ask them to highlight gaps or misconceptions instead of giving you the full solution. Compare their answers. Revise their feedback using *Thinking in Java* Chapter 1 before accepting suggestions.

[assignment_02_06] – Identify the Object Roles

```
class Sensor {
    private double reading = 0.0;
    private long updateRate = 12; //updates each 12 ms
    private String LastError = "";

    public void calibrate(double offset) {
        reading = Math.max(0.0, reading + offset);
    }

    public double getReading() {
        return reading;
    }
}
```

- Describe which methods are mutators and which are inspectors, and justify how the design follows the "mainly inspection and mutation" guideline.
- Add at least one more inspector and one more mutator, and a constructor.
- Use two different LLMs to review your code. Ask it to try an criticize and generate examples that break it(s logic) and double-check against the JavaDoc for `Math.max`. Do you agree/disagree? Why?

[assignment_02_07] – Tiny Library Tour

- Investigate `DateTimeFormatter` and `DateTimeFormatterBuilder` from `java.time.format`. Summarise the role of each class and how they complement one another.
- Start from the official Java API docs and draft your own description plus at least one formatting pattern you would configure manually.
- After drafting your notes, ask at least two LLMs to explain both classes by generating three distinct example use cases (include expected input and output sketches). Compare the suggestions with the docs and adjust your write-up only when you can justify every change.

[assignment_02_08] – Constructor Practice

- Write a `Notebook` class with: fields for `title`, `pageCount`, a constructor that enforces a minimum page count of 16, and methods `rename(String)` and `pages()` aligned with the inspector/mutator rule.
- Write the class outline manually or in your IDE without auto-completion. Compile to ensure your code is syntactically correct.
- After finishing, show an LLM your code and ask it for edge cases you should test. Update your code accordingly.