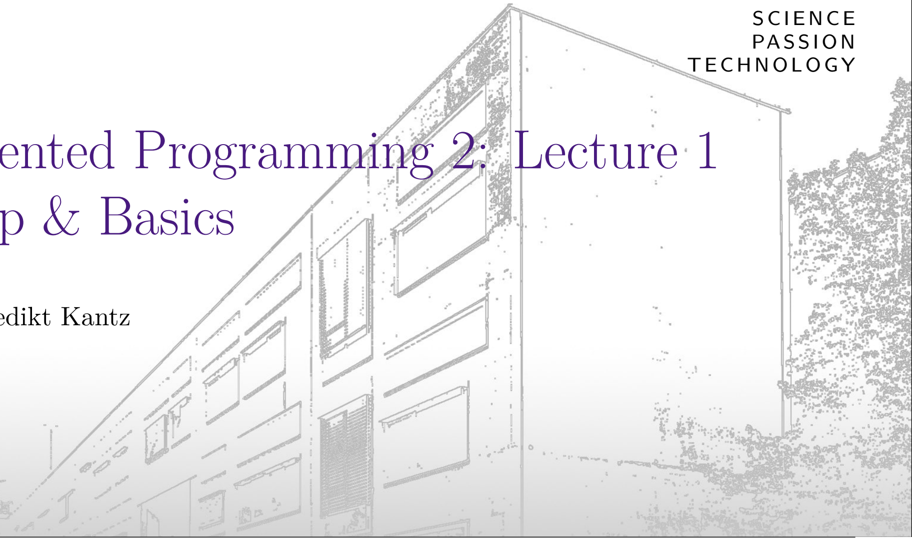
SCIENCE
PASSION
TECHNOLOGY



Object-Oriented Programming 2: Lecture 1
OOP Recap & Basics

Tobias Schreck, Benedikt Kantz

Slido for Q&A

STOP DOING OOP

- MEMORY WAS NOT MEANT TO BE LAID OUT IN OBJECTS

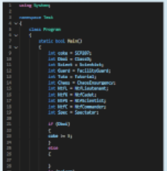
- YEARS of objects yet NO ADVANTAGES FOUND OVER STRUCTS

- Want to store various types of data in the same place? We had a tool for that: it was called "void*"

- "Yeah, I need to stop using references to objects in my code, its causing memory leaks" -- Statements dreamed up by the utterly deranged

LOOK what the C# and JavaScript developers have been demanding wages for

(This is REAL CODE, written by REAL devs):



```
using System;

namespace Test
{
    class Program
    {
        static bool Main()
        {
            int code = SCP009;
            int DBoi = ClassD;
            int Scient = Scientist;
            int Guard = FacilityGuard;
            int Tuto = Tutorial;
            int Chaos = ChaosInsurgency;
            int NtfL = NtfLieutenant;
            int NtfN = NtfCadet;
            int NtfS = NtfScientist;
            int NtfC = NtfCommander;
            int Spec = Spectator;

            if (DBoi)
            {
            code += 1;
            }
            else
            {

            }
```

Why should we use OOP?

Advantages of OOP

- **Modularity:** Objects can be written and maintained independently
- **Reusability:** Objects can be reused in different contexts
- **Extensibility:** Objects can be extended by inheritance
- **Encapsulation:** Objects hide their internal state

Disadvantages of OOP

- **Complexity:** OOP can be more complex than procedural Programming

OOP1 Recap

OOP1 Recap

- C++: all about simple structures, datatypes, . . .
- You had:
  - Classes, attributes, inheritance
  - Polymorphism
  - Smart Pointers
  - Operator Overloading (not in Java...)
  - Templates

Translating C++ to Java

C++	Java
`int main()`	

Translating C++ to Java

C++	Java
`int main()`	`public static void main(String[] args)`

Translating C++ to Java

C++	Java
`int main()`	`public static void main(String[] args)`
`class MyClass`	

Translating C++ to Java

C++	Java
`int main()`	`public static void main(String[] args)`
`class MyClass`	`class MyClass`

Translating C++ to Java

C++	Java
`int main()`	`public static void main(String[] args)`
`class MyClass`	`class MyClass`
`private: \n`	

Translating C++ to Java

C++	Java
`int main()`	`public static void main(String[] args)`
`class MyClass`	`class MyClass`
`private: \n`	`private <type>`

Translating C++ to Java

C++	Java
`int main()`	`public static void main(String[] args)`
`class MyClass`	`class MyClass`
`private: \n`	`private <type>`
`public: \n`	

Translating C++ to Java

C++	Java
`int main()`	`public static void main(String[] args)`
`class MyClass`	`class MyClass`
`private: \n`	`private <type>`
`public: \n`	`public <type>`

Translating C++ to Java

C++	Java
`int main()`	`public static void main(String[] args)`
`class MyClass`	`class MyClass`
`private: \n`	`private <type>`
`public: \n`	`public <type>`

Translating C++ to Java

C++	Java
`int main()`	`public static void main(String[] args)`
`class MyClass`	`class MyClass`
`private: \n`	`private <type>`
`public: \n`	`public <type>`

Translating C++ to Java

C++	Java
`int main()`	`public static void main(String[] args)`
`class MyClass`	`class MyClass`
`private: \n`	`private <type>`
`public: \n`	`public <type>`
`std::cout << "Hello";`	

Translating C++ to Java

C++	Java
`int main()`	`public static void main(String[] args)`
`class MyClass`	`class MyClass`
`private: \n`	`private <type>`
`public: \n`	`public <type>`
`std::cout << "Hello";`	`System.out.println("Hello");`

Introduction to Java ♨ I

- The Java programming language
  - Introduced 1996 (James Gosling lead, Sun Microsystems, now Oracle)
- Some main ideas behind Java
  - Platform independence: Write once, run everywhere
  - Wide applicability: from embedded to desktop, server and compute clusters
  - Rich set of class libraries

Introduction to Java ☕ II

- Easy entry for C/C++ programmers with less low-level concerns for programmer
  - "Safe" language: Immune in absence of native methods to buffer overruns, array overruns, wild pointers or memory corruption
- One of the most widely used programming languages from introduction to today
  - THE language in many enterprise systems
  - Wide use also in mobile, distributed and embedded systems

Why ☕

- Disclaimers:
  - There is no single best programming language
  - Generally: Broaden your horizon of languages
  - Obtain programming competence by doing it
- Comparable in speed to natively compiled code (just in time compiler)
- Rich API support included
  - Doing / learning networking, parallelism, graphics, GUI, etc. is straightforward

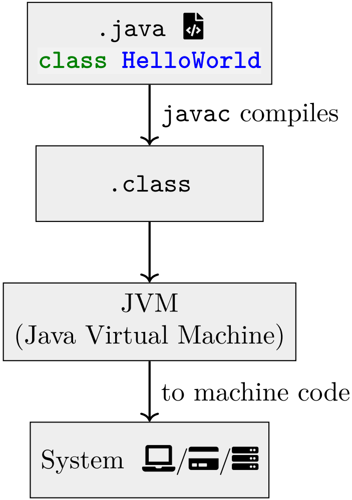
You first ☕ program

- Main components of a Java program:
  - Package declaration (optional)
  - Import statements
  - Class definition with main() method

```java
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
public class HelloWorld {
    private static final Logger logger =
    ↪  LogManager.getLogger(HelloWorld.class);
    public static void main(String[] args) {
        logger.info("Hello, World!");
    }
}
```

The Java Stack: From Source to Bytecode

- Java source (`.java`) is compiled by `javac` into a `.class`.
- Bytecode is executed by the JVM on any supported platform.



`.java`

`class HelloWorld`

`javac` compiles

`.class`

JVM
(Java Virtual Machine)

to machine code

System 💻/💳/🗄

Java JDK vs JRE

- **JDK (Java Development Kit):**
  - Full-featured software development kit for Java.
  - Includes the JRE, compiler (`javac`), debugger, and development tools.
- **JRE (Java Runtime Environment):**
  - Provides libraries, Java Virtual Machine (JVM), and other components to run Java applications.
  - Does *not* include development tools (e.g., compiler).
- **Summary:** `JDK = JRE + Development Tools`

Classes and Objects

- Class: Blueprint for creating objects.
- Object: Instance of a class created using the `new` keyword.

```java
public class Car {
    String color;
    public Car(String color) {
        this.color = color;
    }
}
Car myCar = new Car("Red");
```

Inheritance

- Allows a class (subclass) to inherit methods and properties from another class (superclass).
- Use the `extends` keyword.

```java
class Animal {
    void eat() {
        System.out.println("Eating...");
    }
}
class Dog extends Animal {
    void bark() {
        System.out.println("Woof!");
    }
}
```

Encapsulation

- Protects object data by using private fields and public getters/setters.

```java
class Person {
    private String name;
    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
}
```

Polymorphism

- Ability of a method to perform different tasks based on the object (**Overriding**).
- C++: `virtual` keyword

```
class Animal {
    void sound() {
        System.out.println("Animal sound");
    }
}
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Bark");
    }
}
```

Abstraction

- Hides implementation details from the user.
- Achieved through abstract classes and/or interfaces.
- C++: pure virtual methods

```java
abstract class Shape {
    abstract void draw();
}
class Circle extends Shape {
    void draw() {
        System.out.println("Drawing Circle");
    }
}
```

Interfaces in ☕

- An **interface** defines a contract of methods that implementing classes must provide.
- Interfaces contain method signatures (no implementation[a]).
- A class uses the `implements` keyword to adopt an interface.

```
interface Drawable {
    void draw();
}

class Rectangle implements Drawable {
    public void draw() {
        System.out.println("Drawing
         ↪  Rectangle");
    }
}
```

[a] usually, modern Java allows it though...

Nested Classes in ☕

- Java allows classes to be defined within other classes (nested classes).
- Types: **static nested classes** and **inner classes** (non-static).
- Useful for logically grouping classes and increasing encapsulation.

Static Nested Class

```java
class Outer {
    static class StaticNested {
        void display() {
            // Cannot access Outer.this or outer fields directly
            System.out.println("Static nested class");
        }
    }
}
```

Nested Classes in ☕

Inner class

- Java allows classes to be defined within other classes (nested classes).
- Types: **static nested classes** and **inner classes** (non-static).
- Useful for logically grouping classes and increasing encapsulation.

```
class Outer {
    private int data = 42;
    class Inner {
        void display() {
            // Can access outer class
            ↪  data using Outer.this
            System.out.println("Inner
            ↪  class, data = " +
            ↪  Outer.this.data);
        }
    }
}
```

Access Modifiers in Java

Modifier	Class	Package	Subclass	World
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	×
(no modifier)	✓	✓	×	×
`private`	✓	×	×	×

Table: Access Modifiers in Java: Visibility from different contexts

☕ vs. C++ Differences

- Memory Management: Java has automatic garbage collection, C++ requires manual management.
- Inheritance: Java supports single inheritance, C++ supports multiple inheritance.
- Pointers: Java does not use explicit pointers.
- Platform Independence: Java programs run on the JVM (which has support for a wide range of platforms - from printers to ARM servers).

Interfaces: Solution to Multiple Inheritance

- Java does not support multiple inheritance with classes.

- Interfaces allow a class to implement multiple types.

```java
interface Flyable {
    void fly();
}
interface Swimmable {
    void swim();
}
class Duck implements Flyable, Swimmable {
    public void fly() {
        System.out.println("Duck flies");
    }
    public void swim() {
        System.out.println("Duck swims");
    }
}
```

Exception Handling in ☕

- Use `try`, `catch`, and `finally` blocks.

```java
try {
    int result = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("Cannot divide by zero");
} finally {
    System.out.println("Execution complete");
}
```

The `throws` Keyword[a]

- In Java, the `throws` keyword is used in method signatures to declare that a method may throw certain checked exceptions.
- This informs callers that they must handle or further declare these exceptions.
- The handling checked at *compile* time!

```java
public void readFile(String path) throws IOException {
    // code that may throw IOException
}
```

[a] Something similar *was* available in C++

Templates (C++) vs Generics (☕)

C++ Templates

- Enable functions/classes to operate with generic types & *values*.

```
template <class T, int max> int
↪ arrMin(T arr[], int n)
{
    int m = max;
    for (int i = 0; i < n; i++)
        if (arr[i] < m)
            m = arr[i];
    return m;
}
```

Java Generics

- Provide type safety for collections and classes – only for *types*!

```
class Box<T> {
    private T value;
    public void set(T value) {
     ↪  this.value = value; }
    public T get() { return value; }
}
Box<Integer> intBox = new Box<>();
intBox.set(123);
```

Collections Overview

- Generics power the collections in Java (similar to the `std::list`) . . . .
- Main interfaces: `List`, `Set`, `Map`

List: Ordered, allows duplicates

```
import java.util.List;
import java.util.ArrayList;

List<String> names = new ArrayList<>();
names.add("Alice");
names.add("Bob");
System.out.println(names.get(0)); // Alice
```

Collections Overview

- Generics power the collections in Java (similar to the `std::list`) . . . .
- Main interfaces: `List`, `Set`, `Map`

Set: Unordered, no duplicates

```java
import java.util.Set;
import java.util.HashSet;

Set<String> uniqueNames = new HashSet<>();
uniqueNames.add("Alice");
uniqueNames.add("Bob");
uniqueNames.add("Alice"); // Duplicate ignored
System.out.println(uniqueNames.size()); // 2
```

Collections Overview

- Generics power the collections in Java (similar to the `std::list`) . . . .
- Main interfaces: `List`, `Set`, `Map`

Map: Key-value pairs

```java
import java.util.Map;
import java.util.HashMap;

Map<String, Integer> ages = new HashMap<>();
ages.put("Alice", 30);
ages.put("Bob", 25);
System.out.println(ages.get("Alice")); // 30
```

Reflection: Listing Methods of a Class

- Java reflection can be used to inspect methods of a class at runtime.
- Example: Print all method names of a class.

```
class Parent{
    protected boo(){}
}
class Example {
    public void foo() {}
    private int bar(int x) { return x; }
}

for (Method m :
↪   Example.class.getDeclaredMethods())
↪   {
    logger.info(m.getName()); // Output:
     ↪  foo, bar
}
```

Annotations in ☕

- **Annotations** provide metadata about code to the compiler or runtime.
- Common built-in annotations: `@Override`, `@Deprecated`, `@SuppressWarnings`
- Custom annotations can be defined for frameworks and tools.

```java
@Override
public String toString() {
    return "Hello";
}
```

Defining Custom Annotations

- Define with `@interface` keyword.
- Can specify elements (like parameters).

```java
@interface MyAnnotation {
    String value();
}

@MyAnnotation("example")
class Demo { }
```

Reflection: Using `.getAnnotation` and Custom Annotations

- Java reflection allows inspecting classes, fields, and annotations at runtime.
- Example: Mark fields for database storage using a custom `@DBField` annotation.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
@interface DBField {}
class User {
    @DBField
    String username;
    int age; // $\times$t marked for DB
}
for (Field field : User.class.getDeclaredFields()) {
    if (field.getAnnotation(DBField.class) != null) {
        logger.warn("Add to DB: " + field.getName());
    }
}
```

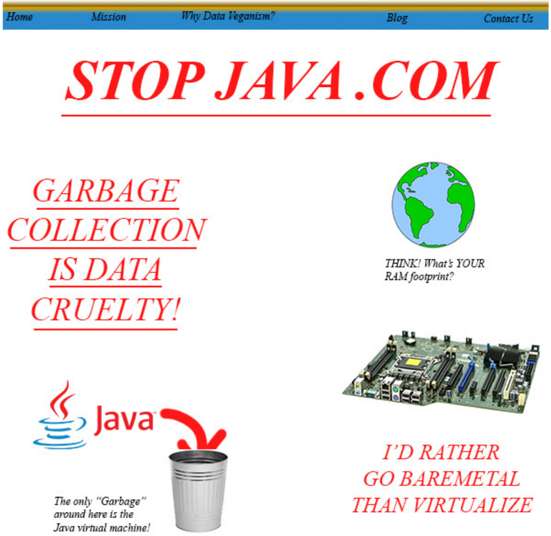
Examples I

- Code examples on Gitlab.
- We will discuss them, looking into different aspect.
- Feel free to expand/go through it at your own pace at home!

Examples II

- Questions:
  - Any further insights for you?
  - What is the difference between `Integer` and `int`?
  - Which data structure would you use for a key-value cache?

Feedback



Home Mission Why Data Veganism? Blog Contact Us

STOP JAVA .COM

GARBAGE COLLECTION IS DATA CRUELTY!

THINK! What's YOUR RAM footprint?

The only "Garbage" around here is the Java virtual machine!

I'D RATHER GO BAREMETAL THAN VIRTUALIZE