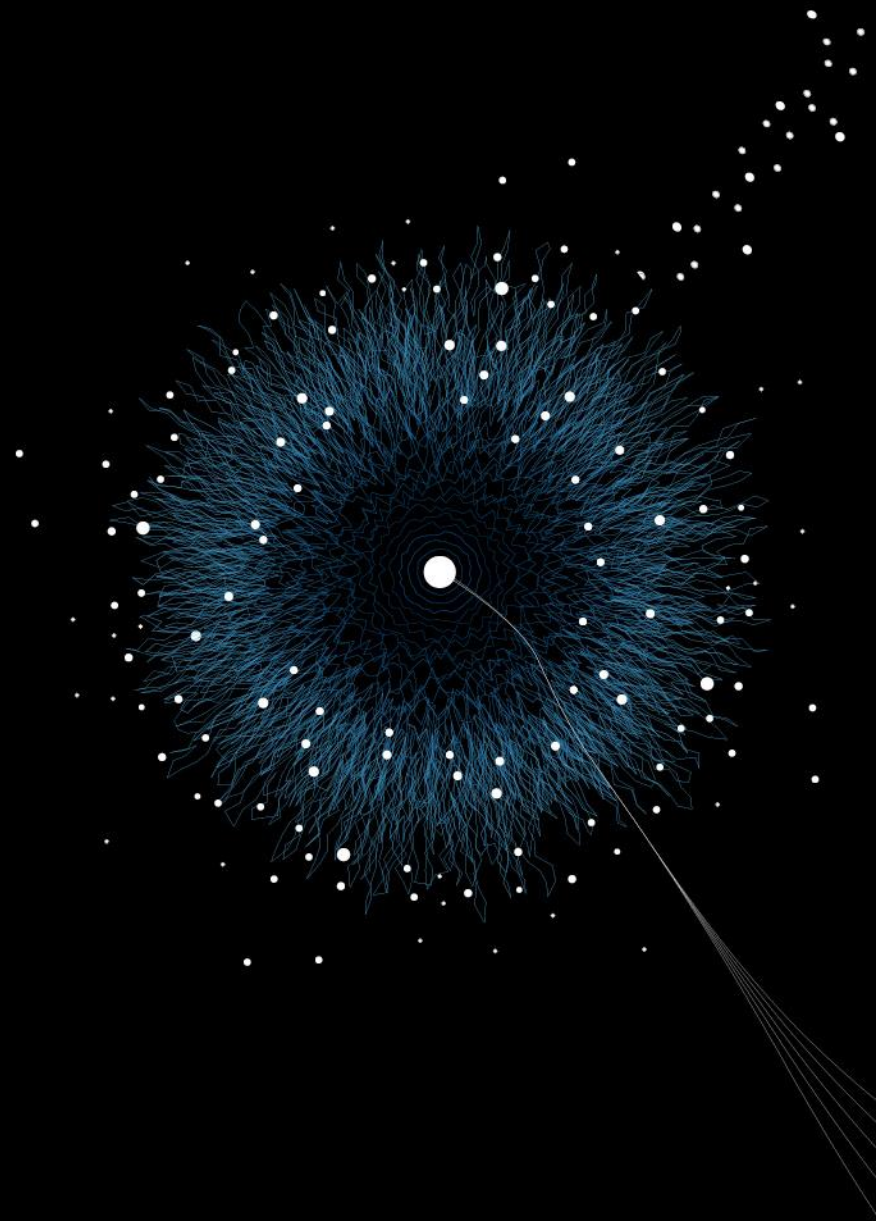
OBJECT-ORIENTED PROGRAMMING 2

M2 PROGRAMMING LECTURE 3.1

BY TOM VAN DIJK & ALEXANDER STEKELENBURG



UNIVERSITY
OF TWENTE.

TOPICS OF THIS LECTURE

- Summary and questions about Week 2
- Overview of object-oriented programming 2
- Object-oriented programming 2 principles
- Quiz

PROGRAMMING OVERVIEW

Week 1 (IP) Procedural programming Arrays Debugging	**Week 2 (IP)** Classes and Objects Documenting and Testing Exceptions (AP)	**Week 3 (IP)** Interfaces, Inheritance, Polymorphism Subtyping Streams and Files (AP)
Week 4 (AP) Collections: Set, List, Map Specification with JML	**Week 5 (AP)** Design Patterns User Interfaces	**Week 6 (AP)** Basic Concurrency Project kick-off
Week 7 (AP) Basic Networking Defensive programming Security	**Week 8/9 (Project)** Project	**Week 10 (Project)** Project Exam

PROGRAMMING OVERVIEW

Java Notes book:

- Week 1: Chapter 1 – 4
- Week 2: Chapter 5.1 – 5.4
- Week 3: Chapter 5.5 – 5.8
- Week 4: Chapter 7.1 – 7.3, 8.1 – 8.3, 9.1 – 9.2, 10.1 – 10.4, 11.1 – 11.2
- Week 6: Chapter 12.1 – 12.3
- Week 7: Chapter 11.4

SO FAR

Good programs are

- Functional
- Reliable, testable
- Understandable, documented
- Formally specified / verified
- Designed for **change**

SO FAR

- Procedural programming (variables, types, procedures, control statements)
- Basics of object-oriented programming (classes, packages, static, final, public, private, constructors, standard methods)
- Program design principles (encapsulation, single responsibility principle, cohesion, coupling, command-query separation)
- Testing and code readability
- Exceptions (Advanced Programming)

Q&A

Any questions about…

- procedural programming
- computational thinking
- object-oriented programming 1
- testing and code readability / documentation
- exceptions

STRUCTURE

- Variables and types **structure** memory
- Procedures **structure** code
- Classes and packages **structure** variables and procedures
- Inheritance **structures** related classes

INHERITANCE

- Every class **extends** another class (by default the Object class)
- This creates a **class hierarchy**
- **Subclasses** extend **superclasses**
- Subclasses **reuse** fields and methods from their superclasses
- From horizontal **has-a** relationships to vertical **is-a** relationships

INHERITANCE

- Subclass gets all fields, but cannot access them (**private**) (*except fields that are **protected**)*
- Subclass gets all methods, but can only access **non-private** methods
- Memory is allocated for all fields, superclass is initialised before subclass
- Subclass constructor first calls superclass constructor **super(…)**

INHERITANCE EXAMPLE

- Class **Pet**
  - name, age, weight, diet
- Class **Dog** **extends** **Pet**
  - breed, trainability, needsLeash, isWorkingDog
- Class **Cat** **extends** **Pet**
  - indoorOnly, likesBoxes, furPattern, usesLitterbox
- Cat **is-a** Pet
- Dog **is-a** Pet

INHERITANCE EXAMPLE

```
class Pet {
    private String name;
    protected Pet(String name) { name = name; }
    public String getName() { return name; }
}

class Dog extends Pet {
    public Dog() {}
    public Dog(String name) { super(name); }
}

Pet theNewPet = new Dog("Butler");
System.out.println("The new pet is " + theNewPet);
```

INTERFACES AND ABSTRACT CLASSES

- An **abstract method** has no **method body**
- If any method is abstract, then the class must be an **abstract class**

```
abstract class Pet {
    abstract void makeSound();
}
```

- An interface only has abstract methods and no fields
- Since Java 8: **default** method implementations in interfaces
  - No access to state (because no fields)
- Classes **extend** a single superclass, but may implement **multiple** interfaces

INHERITANCE EXAMPLE

```
abstract class Pet {
    private String name;
    protected Pet(String name) { this.name = name; }
    public String getName() { return name; }
    abstract public void makeSound();
}

class Dog extends Pet {
    public Dog(String name) { super(name); }
    public void makeSound() { System.out.println("woof!");
}

Pet theNewPet = new Dog("Butler");
System.out.println("The new pet is " + theNewPet);
theNewPet.makeSound();
```

INHERITANCE EXAMPLE

```
abstract class Pet {
    private String name;
    protected Pet(String name) { this.name = name; }
    public String getName() { return name; }
}

interface SoundProducer {
    void makeSound();
}

class Dog extends Pet implements SoundProducer {
    Dog(String name) { super(name); }
    void makeSound() { System.out.println("woof!");
}
```

INHERITANCE

- When to use?
  - **Extend** and **reuse** a class
  - Provide multiple **alternative implementations**
    - Usually use an interface (or abstract class)

- When to not use?
  - When **composition** is cleaner than inheritance
  - **Danger**!! Tight coupling!

FUNCTIONAL INTERFACES

- Define a contract / functionality that a class must fulfill
- Specifies **what** a class should do, not **how** it does it
- Documentation (Javadoc) extremely important
- Classes can (and often do) implement multiple interfaces

Why?

- Lower coupling: reduces dependency on specific implementations
- Cleaner interface between packages / modules of a program
- Flexibility: easy to change implementation without changing dependent code
- Testing: easier to test by providing **mock implementations** of interfaces
- **Downside**: too many / unnecessary interfaces clutter code

POLYMORPHISM

- Week 2: **Overloading** (static polymorphism, early binding)
  - same method name, different method signature
- Week 3: **Overriding** (dynamic polymorphism, late binding)
  - same method signature, subclass with different method body
  - cannot override **private** or **final**
- **Be careful** with overloading, avoid confusion…

TYPING AND SUBTYPING

- **Dynamic** type: the run-time / real type of an object, determined at construction
  - **instanceof** checks the real type
  - Example: “a instanceof B” checks if the real type of a is a subtype of B
  - A **dynamic cast** explicitly converts **to a subtype**

- **Static** type: the type of a variable that references an object
  - Checked by the compiler when assigning to variables/parameters
  - A **static cast** automatically converts **to a supertype**

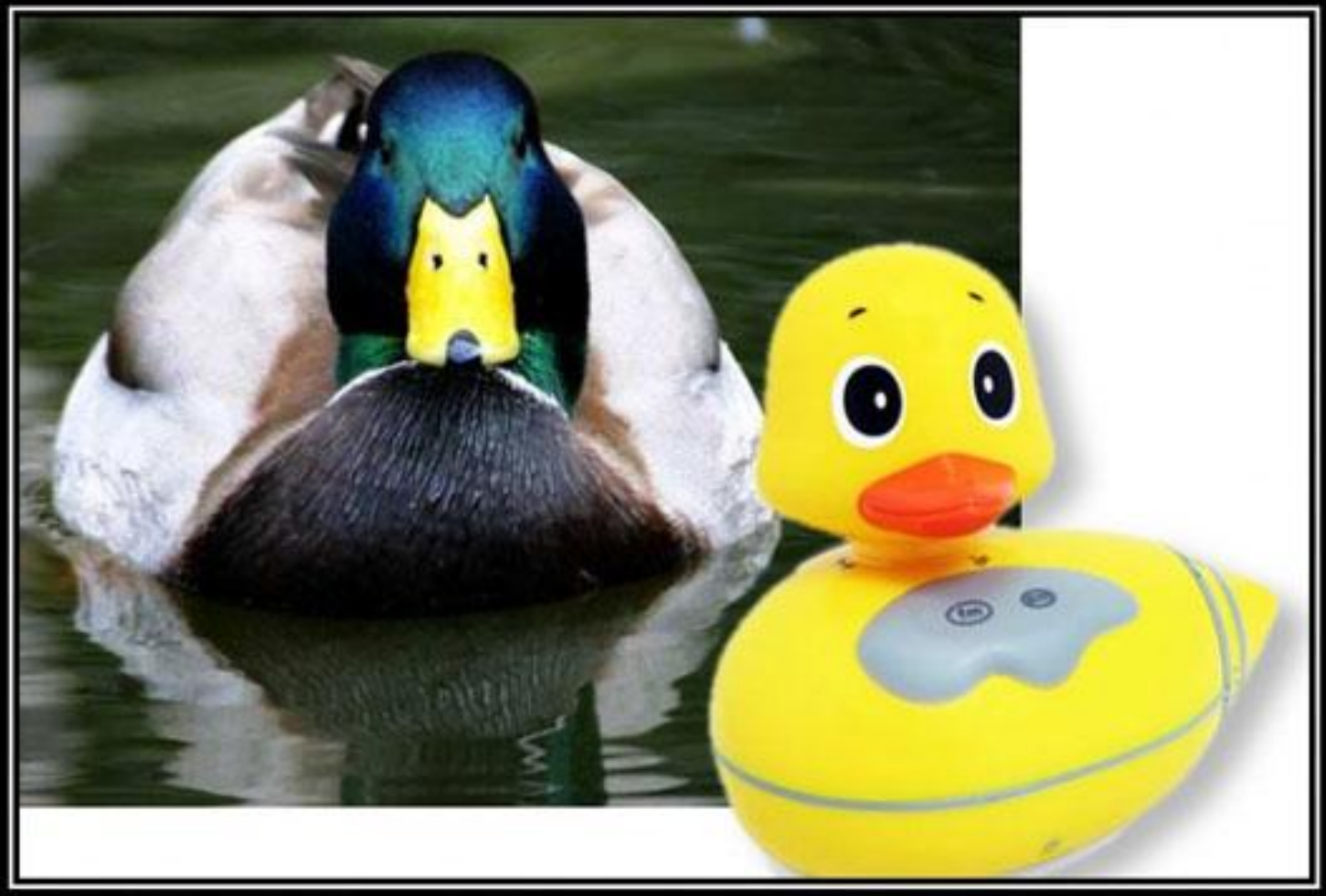
- `A a = (B)c;`

  has a dynamic cast (c to type B) and an (implicit) static cast (type B to type A)

LISKOV SUBSTITUTION PRINCIPLE

- If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program

- Example
  - Duck superclass has a fly() method
  - Implement subclass RubberDuck….
  - **Antipattern**: disabling expected behavior in subclasses



LISKOV SUBSTITUTION PRINCIPLE

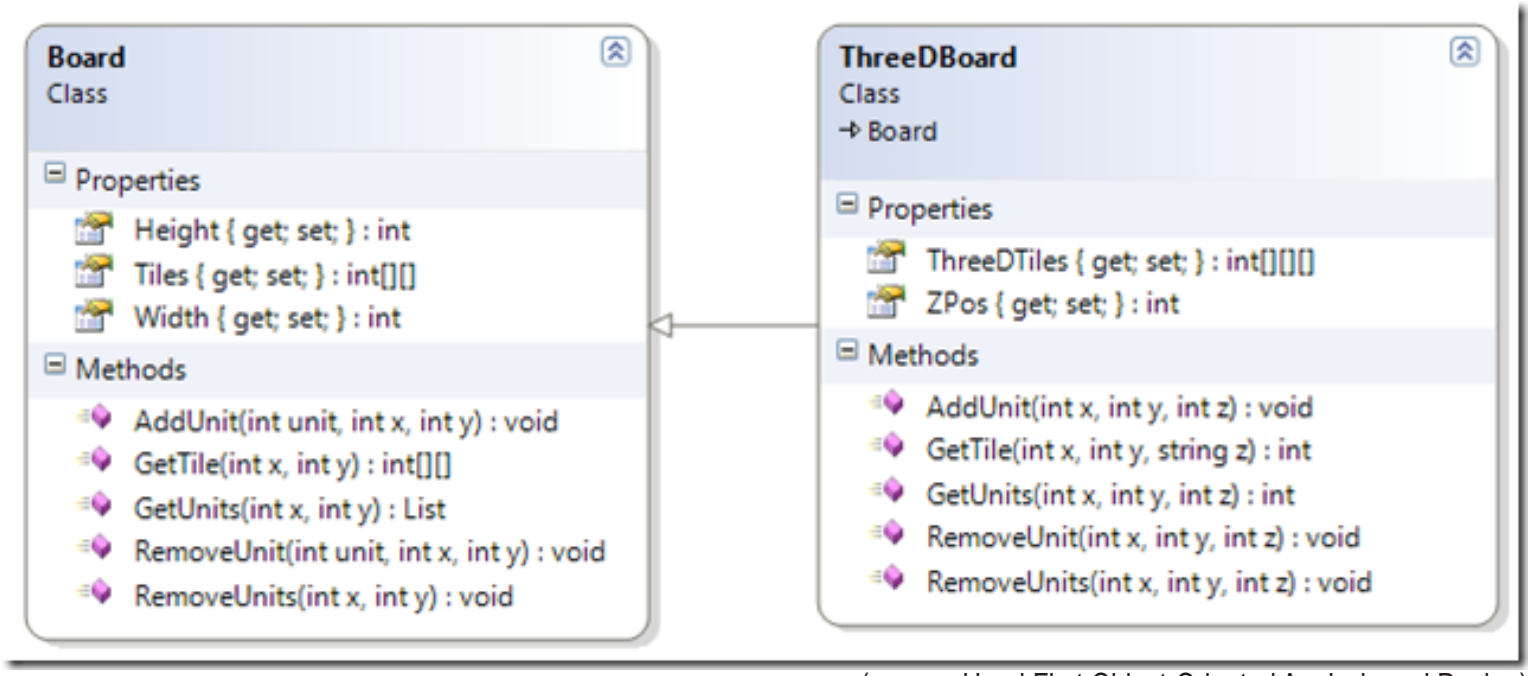
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

COMPOSITION VS INHERITANCE

- **Composition**: reuse by assembling existing components (via **has-a** relationships)
- **Inheritance**: reuse by inheriting existing functionality

- Composition often offers more flexibility
- Some languages (Go, Rust) don't even offer inheritance

- Downsides to composition over inheritance:
  - Need **forwarding** methods (to "delegate")



(source: Head First Object-Oriented Analysis and Design)

- ThreeDBoard extends Board
- Methods in Board still assume 2D coordinates
- Solution: composition!!

OPEN/CLOSED PRINCIPLE

- Software entities (classes, methods) should be open for extension but closed for modification
- Design classes that can be extended (through inheritance or composition) without modifying their core implementation
- Design for change; minimize change propagation
- Works well with the Single Responsibility Principle

INTERFACE SEGREGATION PRINCIPLE

- Avoid forcing a class to implement interfaces it doesn’t use (low coherence)
- Break down a bloated interface into more focused ones
- Avoid unnecessary coupling and dependencies

DEPENDENCY INVERSION PRINCIPLE

- “High-level modules should not depend on low-level modules”
  - Example: handling an order should not depend on all the details of database handling classes, email sending classes, etc.
- Classes should use abstractions (interfaces) instead of implementations
- Constructors get objects they need by interface instead of calling the constructor
- Easier to unit test (via mock implementations)
- Easier to extend (add new implementations of interfaces) and configure
- Example: Frameworks like Spring Boot

```java
public class FileLogger {
    public void log(String message) {
        // Writes message to a file
    }
}

public class Application {
    private FileLogger logger = new FileLogger();

    public void process() {
        // Application logic
        logger.log("Process started");
    }
}
```

```java
public interface Logger {
    void log(String message);
}

public class FileLogger implements Logger {
    public void log(String message) {
        // Writes message to a file
    }
}

public class Application {
    private Logger logger;

    public Application(Logger logger) {
        this.logger = logger;
    }

    public void process() {
        // Application logic
        logger.log("Process started");
    }
}
```

THE SOLID DESIGN PRINCIPLES

- **S**ingle Responsibility Principle
- **O**pen/Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

- Don’t worry too much about it if this is your first programming experience!

MORE PRINCIPLES

- **DRY**: Don’t Repeat Yourself
  - Avoid code duplication, instead reuse!
  - Redundant code leads to bugs!
- **KISS**: Keep It Simple, Stupid
  - Avoid unnecessary complexity
- **YAGNI**: You Aren’t Gonna Need It
  - Don’t build features “just in case we’re gonne need it later”

OVERVIEW OF WEEK 3 EXERCISES

- 1 – 6 Password checker exercise (basic inheritance)
- 7 – 10 Bill and Bill printer (using interfaces, abstact classes, nested classes)
- 11 – 16 Password-protected Hotel safe
- 17 – 18 Updated Hotel TUI

SUMMARY SO FAR

The basics of object-oriented programming in Java, part 2

- Inheritance
- Polymorphism
- Typing and subtyping, dynamic (runtime) vs static (reference) types
- Abstract classes and interfaces

Solid program design principles

- **S**ingle Responsibility Principle
- **O**pen/Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Any questions about **object-oriented programming 2**?

QUESTION

```
public class Date {
    private int day;
    private int month;
    private int year;

    public int getDay() {
        return day;
    }

    public int getMonth() {
        return month;
    }

    public int getYear() {
        return year;
    }

    // ...
}
```

What sort of variable is *day*?

A. Local variable
B. Parameter
C. Instance variable
D. Static variable
E. Constant

QUESTION

```
public class A {
   int i = 10;
   public void method(double i) {
      method((int) i);
   }
   public void method(int j) {
      System.out.println(j);
   }
   public void method() {
      method(i);
   }
   public static void main(String[] args) {
      A a = new A();
      a.method(5.0);
      a.method();
   }
}
```

What will be printed?

A. Prints 5, then 10
B. Prints 10, then 5
C. Prints 5 twice
D. Compiler error
E. Run-time error

QUESTION

```
public class Counter {
    private int counter;

    public int getCounter() {
        return counter;
    }

    public void incCounter() {
        counter++;
    }

    public static void main(String[] args) {
        Counter c = new Counter();
        System.out.println(Counter.getCounter());
    }
}
```

What will happen if you run main?

A. 0
B. 1
C. Compiler error
D. Run-time error

QUESTION

```
public class Box {
    private int height;
    private int length;

    public Box (int h, int l) {
        height = h;
        length = l;
    }

    public static void main(String[] args) {
        Box b = new Box(“42”, “3”);
    }
}
```

What will happen if you run main?

A. Compiler error
B. Run-time error
C. Box with height = 42, length = 3
D. Box with height = 2, length = 1

QUESTION

```
public class Item {
    public int x;

    private void update(int n, boolean flag) {
        if (flag) {
            n = x * n;
        }
        return x == n;
    }
}
```

What problem will the compiler detect?

A. Public instance variable
B. Private method
C. Result type of update
D. Method changes parameter

QUESTION

```
public class Print {
    public static int a;

    public static void main(String[] args) {
        int b;

        System.out.println("a : "+a);
        System.out.println("b : "+b);
    }
}
```

What will happen?

A. Compiler error
B. Run-time error
C. Nothing
D. “a : 0” and “b : 0” will be printed

QUESTION

```
public class Counter {
    private int counter;
    public void incCounter() { counter++; }
}

public class Foo {
    public static void main (String [] args) {
        Counter c = new Counter();

        c.incCounter();
        c.incCounter();
        c.incCounter();

        System.out.println(c.counter);
    }
}
```

What will happen?

A. 4
B. 3
C. Run-time error
D. Compiler error

QUESTION

```
public class Counter {
    private int count;

    public Counter(int count) {
        count = count;
    }

    public int getCount() {
        return count;
    }

    public static void main(String[] args) {
        Counter c = new Counter(4);
        System.out.println(c.getCount());
    }
}
```

What will be printed on the screen?

A. Undefined
B. 2
C. 4
D. 0

QUESTION

```
public class Item {
    public static int created = 0;

    public Item() {
        created = created + 1;
    }

    public static void main(String[] args) {
        Item i1 = new Item();
        Item i2 = new Item();

        System.out.println(Item.created);
    }
}
```

What will happen?

A. Prints “2”
B. Error: created should be private
C. Error: created belongs to object
D. Error: integer cannot be printed

QUIZ QUESTIONS

A class has a final field `private final int[] x = new int[4];`

- Is it allowed to change the contents like: `x[0] += 5;`

A class has a final field `private final List x = new ArrayList();`

- Is it allowed to change `x` to a different list?
- Is it allowed to change the list by adding or removing items?

QUIZ QUESTIONS

When does Java make a *default constructor* (no parameters, empty body)

A. Always
B. Only if there is no constructor without parameters yet
C. Only if there is no constructor yet
D. Only if all defined constructors are private

QUIZ QUESTIONS

What is the default implementation of `equals` (comparing two objects)?

A. Throw a `NotImplementedError` runtime exception
B. Compare the contents of all fields using `==`
C. Compare the contents of all fields using `equals`
D. Compare the two objects if they are the same reference
E. Always returns `false`