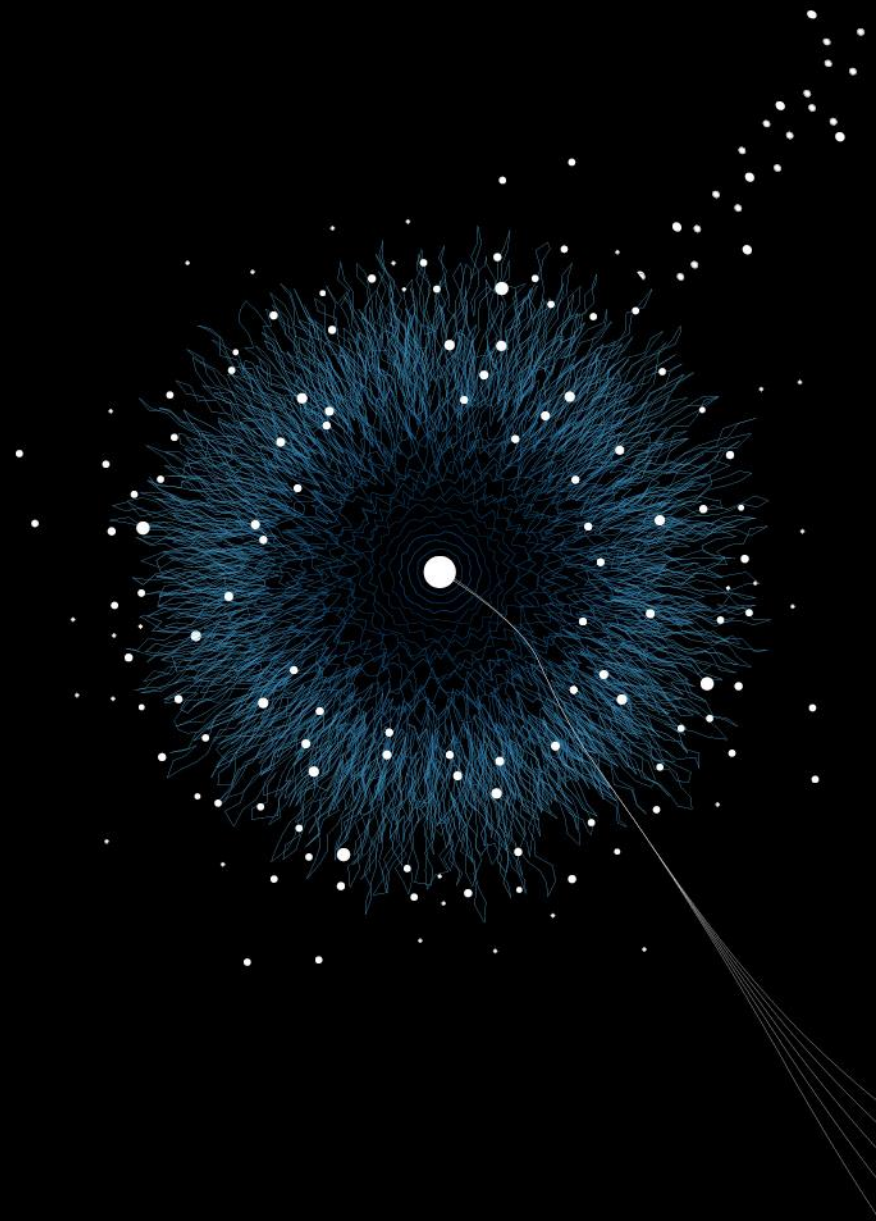
OBJECT-ORIENTED PROGRAMMING 1

M2 PROGRAMMING LECTURE 2.1

BY TOM VAN DIJK



UNIVERSITY
OF TWENTE.

TOPICS OF THIS LECTURE

- Summary and questions about Week 1
- Overview of object-oriented programming 1
- Object-oriented programming principles
- Quiz

PROGRAMMING OVERVIEW

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

- Variables and types **structure** memory
- Procedures **structure** code
- Primitive types (byte, short, int, long, float, double, char, boolean, arrays)
- Control statements: if, else, switch, for, while, do-while, continue, break

Q&A

Any questions about **procedural programming** or **computational thinking**?

STRUCTURE

- Variables and types **structure** memory
- Procedures **structure** code
- Classes and packages **structure** variables and procedures

STRUCTURE

Add more structure with **object-oriented programming**!

- Every **class** has **fields** (class variables) and **methods**
- **Decomposition** and **abstraction**: implementation details of other classes / packages are irrelevant

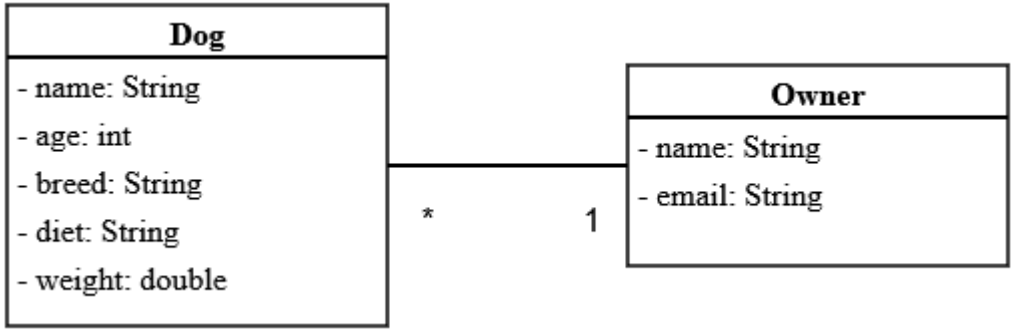
How to apply?

- Separate by **data objects**
- Separate by **functionality**

EXAMPLE: KENNEL

- Class **Dog**
  - Name, age, breed, diet, weight, etc
- The class defines a **blueprint** for all **instances / objects** of that class
- There can be many dogs in the kennel
- Each dog is an instance of the Dog class
- Each **Dog** instance / object has the same fields and methods

EXAMPLE: KENNEL



- Class **Dog**
  - Name, age, breed, diet, weight, etc
- Class **Owner**
  - Name, contact information
- An owner **has-a** dog (maybe multiple); a dog **has-an** owner (only one)
  - Dog has a field **owner** of type **Owner**
  - Owner has a field **dogs** of type **Set<Dog>** or **Dog[]**
- **Reference types**: Owner, Dog, etc, “point at” the location of the object in memory

EXAMPLE: KENNEL

- How would the Kennel class work? (i.e. what **methods** do we need to offer to other classes?)

- How do we get all dogs currently in the kennel?
- How do we check in and check out a dog?
- How do we record information to produce a bill for the owner?
- How do we compute a feeding schedule for the dogs?

- Remark: A class that does everything is called a **God class** – we want to avoid this!

EXAMPLE: KENNEL

Separate by functionality:

- Class **Kennel**
  - registerDog
  - checkIn
  - checkOut
- Class **PaymentSystem**
  - makeBill
  - completePayment
- Class **FeedingSchedule**
  - makeFeedingSchedule

SOME BASICS OF JAVA OOP

We've seen so far:

- Classes have **fields** and **methods**

Next up:

- **Standard** object methods
- The keywords **static** and **final**
- Access keywords **public** and **private**
- Object **lifecycle** (construction and destruction)

STANDARD OBJECT METHODS

Every object in Java always has these methods:

- `String toString()`
  - returns a string representation of the object
  - default: class name + hash code
  - override for readable logging and debugging

- `boolean equals(Object obj)`
  - returns `true` if the object is “equal” to the other
  - default: compare references (so just ==)
  - override to compare content, not references (needed for sets and maps and comparing by value)

- `int hashCode()`
  - returns an integer for use in hash-based collections
  - must be consistent with `equals()`
  - override if `equals()` is overridden

OBJECT KEYWORDS

Common keywords (see classy keywords topic)

- **static** for a field
- **static** for a method
- **final** for a field
- **final** for a method – not in week 2!
- **static final** for a field

OBJECT KEYWORDS

```
class Counter {
    static int count = 0;  // Shared across all instances
    int mynumber;

    Counter() {
        mynumber = count++;
        System.out.println("I am number: “ + mynumber);
    }
    static void printCount() {
        System.out.println("Total created: " + count);
    }
}

Counter a = new Counter();   // Output: I am number: 0
Counter b = new Counter();   // Output: I am number: 1
Counter.printCount();        // Output: Total created: 2
```

MORE OBJECT KEYWORDS

Access modifiers:

- **private**: no other class may access this method or field
- **public**: every class may access this method or field
- **package private**: only classes in the same package may access this method or field
- package private is the default if no keyword is used!

OBJECT LIFECYCLE

How does Java ***make*** a new object?

1. Static initialisation (only once!)
   a) Field initialisation
   b) Static initialiser blocks

2. Allocate memory for the object, all fields get default values (`0, false, null`)

3. Instance initialisation
   1. Field initialisation
   2. Initialiser blocks

4. Run the constructor

Default constructor (if no constructor is defined) is just an empty block

OBJECT LIFECYCLE

```
public class A {
    static int s = initField();                                  // (1)
    int x = initField();                                         // (3)
    static int initField() { return 42; }
    static { System.out.println(“Static instance block”); }      // (2)
    { System.out.println("Instance block"); }                    // (4)
    A() { System.out.println("Constructor"); }                   // (5)
}

A object = new A();   // steps 1, 2, 3, 4, 5
A object2 = new A();  // steps       3, 4, 5
```

OBJECT LIFECYCLE

How does Java ***delete*** an object?

- Called **garbage collection**
- Once an object is unreferenced (no variable or field points to it) it could eventually be deleted
- Don’t worry about it!
- Different in languages like C, C++

OBJECT-ORIENTED PROGRAMMING PRINCIPLES

- Encapsulation
- Single Responsibility Principle
- Cohesion
- Coupling
- Command Query Separation

ENCAPSULATION

- A method of **abstraction**
- A package/class **encapsulates** data and methods: separates internal and external (interface)
- All fields **must** be **private**.
  - Only access fields through methods!
  - Protects internal state against unintended modifications
- Methods should be **private** or **package private** unless intended for public use
  - Controlled access over data and methods
- Other classes do not (should not) need to know how something is implemented!

ENCAPSULATION

- Remember: a good program is not only **functional**, it is also **reliable**, **documented**, designed for future **change**, etc.
- What can go wrong if we only care about functional? Public fields and methods are simple and easy?!
- Class design with private fields and well-chosen public methods: fewer bugs, clearer intended usage

OBJECT-ORIENTED PROGRAMMING

Consider a BlogPost class with the following methods:

- BlogPost(String title, String content)
- String getTitle()
- String getContent()
- long getLength()
- String formatForDisplay()
- void saveToDatabase()
- void sendNotification(String email)
- int calculateReadingTime()

Do you see a potential problem with this class design?

OBJECT-ORIENTED PROGRAMMING

Responsibilities of BlogPost:

- Stores data (fields, getters)
- Presentation (formatForDisplay)
- Persistence (saveToDatabase)
- Communication (sendNotification)
- Analysis (calculateReadingTime)

SINGLE RESPONSIBILITY PRINCIPLE

- **Single responsibility principle**: A class should focus on one role or task in a system
- "A class should have only one reason to change" (Robert C. Martin)
  - … if the presentation changes
  - … if the database model changes
  - … if the communication method changes
  - … if the reading speed changes
- Why apply the single responsibility principle?
  - Easier to test
  - More reusable without unrelated other responsibilities
  - Changes are safer with fewer unintended side effects
  - Clearer code

COHESION

Cohesion describes how strongly the parts of a class are related and how closely they work together

- **High cohesion**: Most methods use most fields, and methods often call each other or rely on shared internal logic.
- **Low cohesion**: Methods use different subsets of fields, rarely interact, and support unrelated functionality

Cohesion and the single responsibility principle: correlated but not identical:

- A class with one responsibility can have unrelated methods → low cohesion
- A class can have multiple tightly related responsibilities → high cohesion

COUPLING

Coupling describes how storngly one class depends on or is affected by other classes

- **High coupling**:
  - A class relies heavily on the internal details of other classes.
  - Changes ripple through the system and increase fragility.

- **Low coupling**:
  - A class interacts with few others, through simple, well-defined interfaces
  - Changes in one class rarely impact others.

- Design goal: Minimize coupling to isolate the impact of changes

- More on this when we consider **design patterns**

COMMAND QUERY SEPARATION

- **Queries** answer a question about the current state, return data **without side effects**
- **Commands** perform an action or change state, *usually* return void
- Encourages clear and predictable class methods (easier to reason about)

Example:

```
• void addItem(Item item)
• boolean contains(Item item)
• List<Item> getItems()

• boolean addItem(Item item)
• boolean remove(Item item)
```

CODE AS COMMUNICATION

- Code and class design is **communication**

- Write for reading
- Use meaningful names
- Avoid obscure solutions, simplicity is better (**KISS** principle – keep it simple, stupid)
- Provide clear and concise documentation

- Good code tells a story that others can understand

UNIT TESTING

- **JUnit tests** for **packages** and for **classes**

- Typically a **test** subdirectory with the same package structure
  - Example: **KennelTest** in same package as **Kennel**
  - Could use **package private** methods for testing

- Clear **single responsibilities** make unit testing easier
- **Low coupling** allows isolated testing of components
- **High cohesion** ensures tests align with class purposes

- Testable code is reliable code

OVERVIEW OF WEEK 2 EXERCISES

- 1 — Wallet: fields and methods
- 2 – 7 — ThreeWayLamp: CQS, testing, TUI
- 8 – 19 — Hotel: Javadoc, testing, TUI

SUMMARY

The basics of object-oriented programming in Java

- Classes and packages **structure** variables and procedures
- Keywords **static**, **final**, **public**, **private**
- **Constructors** and **initializers**
- Standard methods **toString**, **equals**, **hashCode**

Solid program design principles

- Encapsulation
- Single responsibility principle, cohesion, coupling
- Command-query separation

Any questions about **object-oriented programming 1**?

FROM PYTHON TO JAVA

Given this Python program, what is the equivalent Java program?

```
def max(x, y):
    if x > y:
        return x
    else:
        return y
```

Python

```
class Max {
    int max (int x, int y)
            if (x > y)
                return x
            else
                return y
}
```

A

```
class Max {
    int max (int x, int y) {
            if (x > y) {
                return x;
            }
            else {
                return y;
            }
    }
}
```

B

```
class Max {
    max (x, y) {
            if (x > y) {
                return x;
            }
            else {
                return y;
            }
    }
}
```

C

What is the type of this expression?

```
int x = 0;
boolean b = false;
double d = 2.0;

(x > 24) || (!b && d + 23 == 15.0)
```

a. int
b. float
c. double
d. boolean

What will be printed on the screen?

```
 public static void main(String [] args) {
        String obj1 = “xyz”;
        String obj2 = “x” + “y” + “z”;
        if(obj1.equals(obj2)) {
            System.out.println("obj1 == obj2 is TRUE");
        } else {
            System.out.println("obj1 == obj2 is FALSE");
        }
}
```

a. Compilation error
b. obj1 == obj2 is TRUE
c. Runtime error
d. obj1 == obj2 is FALSE

What is the value of z?

```
int x = 23;
int y = 34;
int z = (int)  (x++) + (++y);
```

a. 57
b. 56
c. 58
d. 59

What will happen?

```
int x = 29/3;

System.out.println(x);
```

a. Compiler error
b. 9 printed
c. 9.66666666… printed
d. Run-time error

WHAT IS THE VALUE

```
double grade = 8.6;
int alsoGrade = (int)grade;

A.  alsoGrade == 8
B.  alsoGrade == 8.6
C.  alsoGrade == 9
D.  alsoGrade == “8.6”
```

WHAT IS THE VALUE

```
String example = “The year is ” + 2020 + 2;

A.  “The year is ”
B.  “The year is 2020”
C.  “The year is 20202”
D.  “The year is 2022”
E.  true
```

WHAT ARE THE VALUES

```
int justSomeValue = 4;
long justAnotherValue = justSomeValue++;
justSomeValue++;

A.  justSomeValue == 4 && justAnotherValue == 5
B.  justSomeValue == 5 && justAnotherValue == 5
C.  justSomeValue == 6 && justAnotherValue == 4
D.  justSomeValue == 6 && justAnotherValue == 6
```

QUESTION

```java
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`