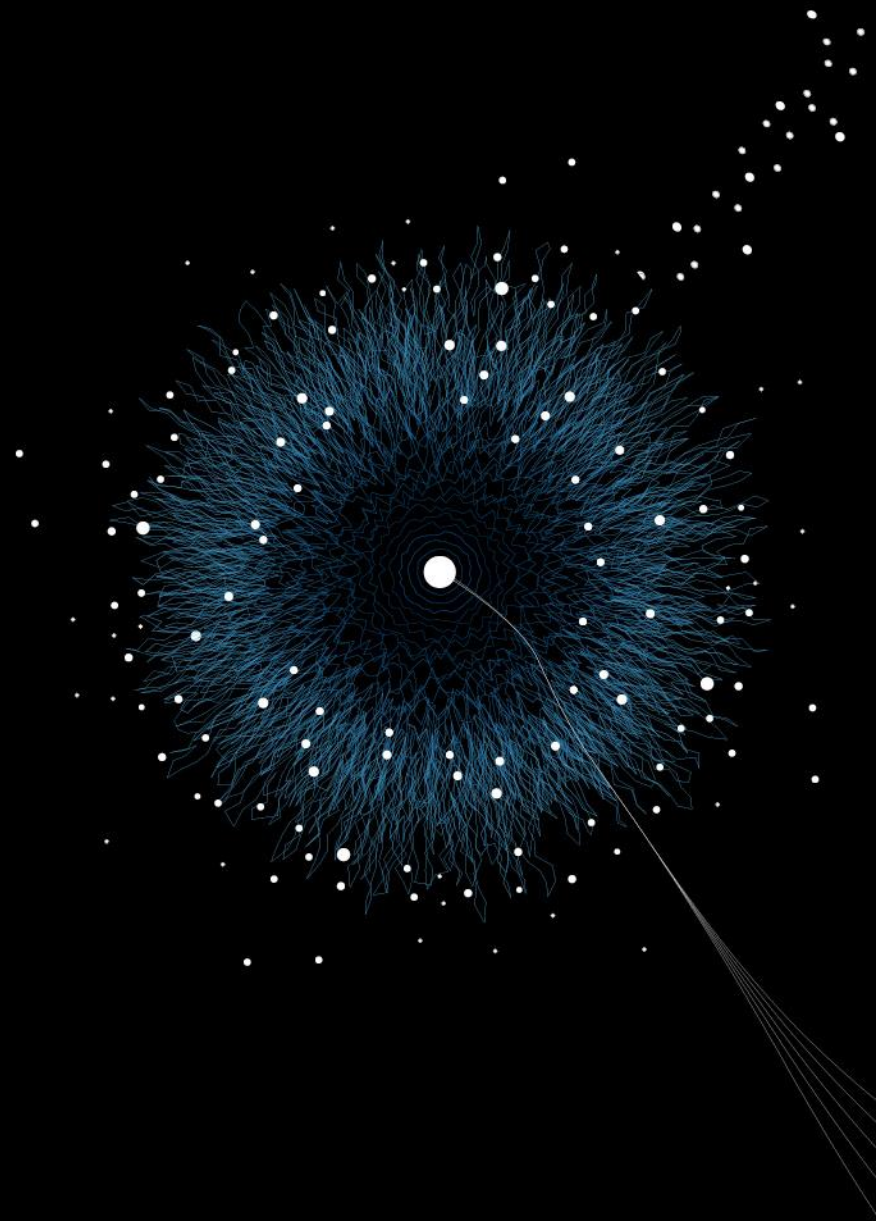
JAVA COLLECTIONS



M2 PROGRAMMING LECTURE 4.1

BY TOM VAN DIJK

UNIVERSITY
OF TWENTE.

TOPICS OF THIS LECTURE

- Summary and questions about Week 3
- Quiz about Week 3
- Overview of Java Collections
- A little bit on computational complexity

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
  - Chapter 7.1 – 7.3 arrays and ArrayList
  - Chapter 8.1 – 8.3 writing correct programs, Exceptions and try…catch
  - Chapter 9.1 – 9.2 recursion, Linked data structures, LinkedList
  - Chapter 10.1 – 10.4 generics, Lists, Sets, Maps, Java Collection Framework
  - Chapter 11.1 – 11.2 I/O streams, Readers, Writers, Files
- 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)
- Inheritance, polymorphism, typing, abstract classes, inheritance
- Streams and files

Q&A

Any questions about…

Introduction to Programming:

- procedural programming
- computational thinking
- object-oriented programming 1 and 2
- testing and code readability / documentation
- the practice exam

Advanced Programming:

- exceptions
- streams and files

QUESTION 1

Which of the following statements is false:

1. Interfaces can only contain abstract methods
2. Abstract classes must contain at least one abstract method
3. Abstract classes cannot have constructors

Which are false?

A. 3
B. 1 and 2
C. 1 and 3
D. 2 and 3
E. 1, 2 and 3

QUESTION 2

Which of the following statements is false:

1. An interface can extend an arbitrary number of other interfaces
2. An abstract class can extend an interface
3. An abstract class can extend a non-abstract class

Which are false?

A. 2
B. 2 and 3
C. 3
D. 1 and 3

QUESTION 3

A is a class and B is another class extending A.
Both have empty constructors.

Which of the following program fragments are wrong:

1. A varA = new B();
2. B varB = new A();
3. B varB = new B(); A varA = varB;

Which are wrong?

A. 2
B. 2 and 3
C. 3
D. 1 and 3

QUESTION 4

```
class Rectangle {
   private int a;
   private int b;
   public void setA(int newA) {
      this.a = newA;
   }
   public void setB(int newB) {
      this.b = newB;
   }
   public int getArea() {
      return a * b;
   }
}
class Square extends Rectangle {
   public void setA(int a) {
      setA(a); setB(a);
   }
   public void setB(int b) {
      setB(b); setA(b);
   }
}
```

```
class Test {
   public static void main(String[] args) {
      test(new Rectangle());
      test(new Square());
   }
   private static void test(Rectangle r) {
      r.setA(3);
      r.setB(4);
      System.out.println("Area: " + r.getArea());
   }
}
```

What does running Test.main() print?

A. “Area: 12” then “Area: 12”

B. “Area: 12” then “Area: 9”

C. “Area: 12” then “Area: 16”

D. “Area: 12”

QUESTION 5

```
class Point {
   boolean equal(Point x) { return false; }
}
class ColorPoint extends Point {
   boolean equal(ColorPoint x) { return true; }
}
class Puzzle {
   public static void main(String[] args) {
      Point p1 = new Point();
      Point p2 = new ColorPoint();
      System.out.println("1: " + p1.equal(p1));
      System.out.println("2: " + p2.equal(p1));
      ColorPoint cp = new ColorPoint();
      System.out.println("3: " + cp.equal(p2));
      System.out.println("4: " + p2.equal(cp));
      System.out.println("5: " + cp.equal(cp));
   }
}
```

Which are true/false?

A. 1: false, 2: true, 3: false, 4: false, 5: true

B. 1: false, 2: false, 3: true, 4: true, 5: true

C. 1: false, 2: false, 3: false, 4: true, 5: true

D. 1: true, 2: false, 3: false, 4: false, 5: true

QUESTION 6

```
1.   interface A {
2.     abstract void gimme();
3.   }
4.
5.   class B extends A {
6.     public static final int value = 0;
7.
8.     int gimme() {
9.        value = value + 1;
10.       return value;
11.    }
12.
13.    abstract int B() {
14.       return gimme() > 0;
15.    }
16.  }
```

Which lines contain errors?

A. 2, 5, 6, 9, 13

B. 2, 6, 9, 14

C. 5, 9, 13, 14

D. 5, 13

QUESTON 7

```
class Puppy {
   public int age = 1;
   public void sayHello() {
      System.out.println(“woof");
   }
}
class SuperDog extends Puppy {
   public int age = 13;
   public void sayHello() {
      System.out.println(“WOOF");
   }
   public static void main(String[] args) {
      SuperDog dog1 = new SuperDog();
      dog1.sayHello();
      System.out.println(dog1.age);
      Puppy dog2 = dog1;
      dog2.sayHello();
      System.out.println(dog2.age);
      System.out.println(dog2 instanceof SuperDog);
   }
}
```

What does running main() print?

A. “WOOF”, 13, “WOOF”, 13, true
B. “WOOF”, 13, “WOOF”, 1, true
C. “woof”, 1, “WOOF”, 13, false
D. “WOOF”, 1, “woof”, 1, true

QUESTION 8

```
public class Thing {
    protected String name;
    protected String owner;
    public Thing(String name, String owner) {
        this.name = name; this.owner = owner;
    }
}
public class Thang extends Thing {
    public Thang(String name, String owner) {
        super(name, owner);
    }
    @Override
    public boolean equals(Object other) {
        if (other == this) return true;
        if (!(other instanceof Thing)) return false;
        return name.equals(((Thing) other).name);
    }
}
Thing a = new Thing(“a thing”, “someone”);
Thing b = new Thang(“a thing”, “someone else”);
Thing c = a;
```

What is true?

A. a.equals(b) && b.equals(a) && !c.equals(a)

B. !a.equals(b) && b.equals(a) && c.equals(a)

C. !a.equals(b) && !b.equals(a) && c.equals(a)

D. None: you get a run-time error

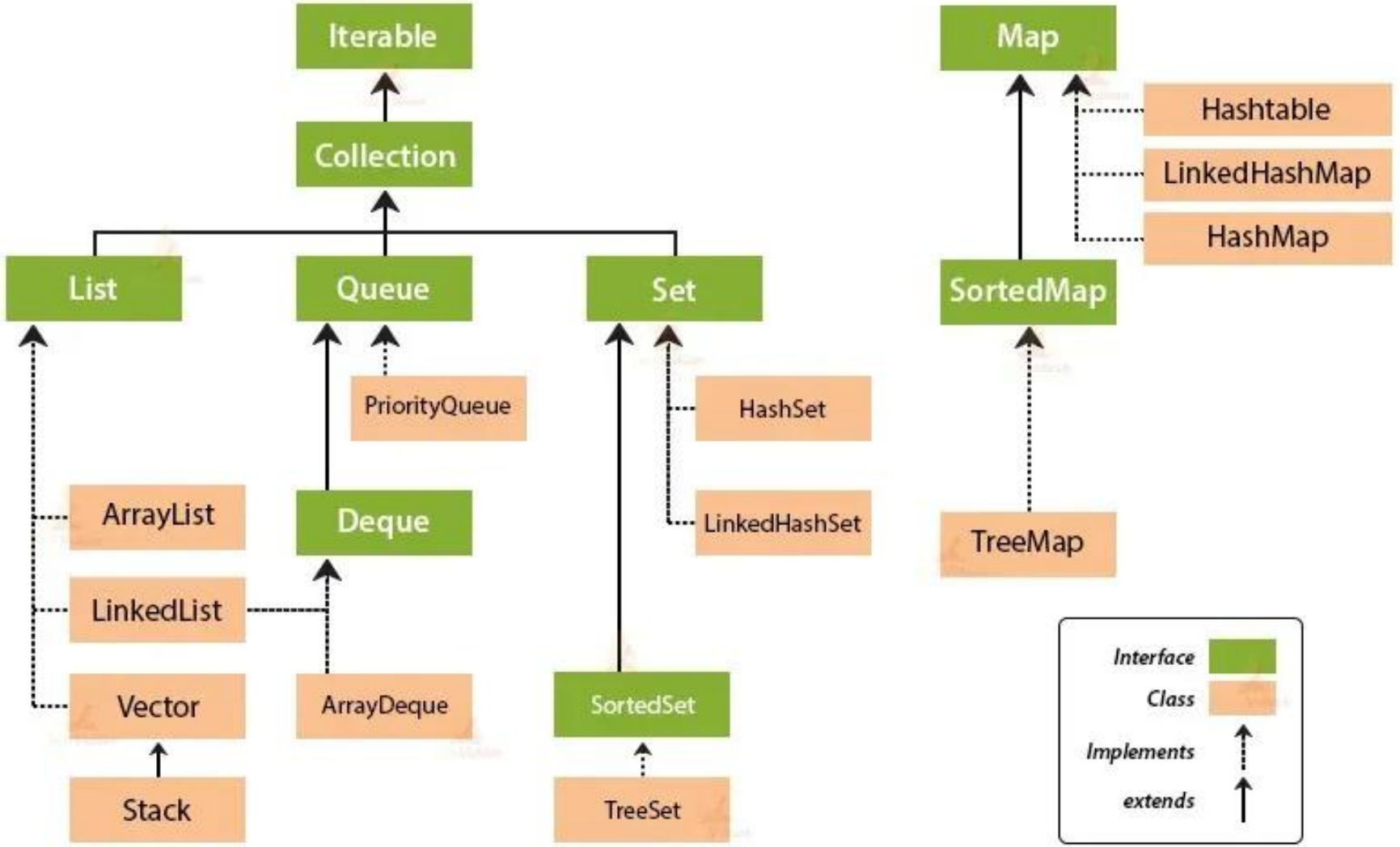
STRUCTURE

- Variables and types **structure** memory
- Procedures **structure** code
- Classes and packages **structure** variables and procedures
- Inheritance **structures** related classes
- Data**structures** and algorithms are core techniques to organise and process data

JAVA COLLECTIONS

- Java Collections library: `java.util` package
- Interfaces: **Collection**, **List**, **Set**, **Map**, and more…
- Implementations: **ArrayList**, **LinkedList**, **HashSet**, **HashMap**, and more…
- Use **generics** type E:
  - List theList → add(Object e), get() returns Object, etc…
  - List<E> theList → add(E e), get() returns E, etc…
  - List<Dog> theDogList → add(Dog e), get() returns Dog, etc…
- Note: primitive values are (currently) not possible with generics
  autoboxing `Integer` for `int`, `List<Integer>`, etc.

Collection Framework Hierarchy in Java



COLLECTION<E>

Base interface for sets and lists

- `size()` returns the number of elements
- `isEmpty()` returns **`true`** if `size() == 0`
- `contains(Object o)` returns **`true`** if the collection contains an object **equal** to o
- `containsAll(collection)` returns **`true`** if the collection contains all elements of the given collection

- `add(E e)` add an element
- `addAll(collection)` add all elements of another collection

- `clear()` remove all elements
- `remove(Object o)` remove one instance of the given object from the collection
- `removeAll(collection)` remove all elements that are in the given collection
- `retainAll(collection)` remove all elements that are not in the given collection

SET<E>

- Inherits all methods of `Collection<E>`
- Ensures that objects are only added **`once`** (according to **`equals`**)
- Two implementations: `HashSet<E>` and `TreeSet<E>`
- `HashSet<E>` uses **`equals`** and **`hashCode`** to determine if objects are equal
- Default choice for collections of objects

LIST<E>

- Stores objects **in a specific order** (``sequence") and **allows duplicates**
- New methods that use the index
  - `add(int index, E element)`
  - `get(int index)`
  - `set(int index, E element)`
  - `remove(int index)`
  - `indexOf(Object o)`
  - `lastIndexOf(Object o)`
  - `sort(Comparator c)`
  - `subList(int fromIndex, int toIndex)`
- Two implementations: `LinkedList<E>` and `ArrayList<E>` (see video topics!)
- Use **if** you need to store duplicates or care about a specific order

MAP<K, V>

• Map is not a Collection!	stores key-value pairs like a dict
• `size()`	returns the number of elements
• `clear()`	remove all elements
• `isEmpty()`	returns **`true`** if `size() == 0`
• `put(K key, V value)`	associate key with value
• `remove(Object key)`	remove a key-value pair (if present)
• `get(Object key)`	get the associated value (or `null`)
• `containsKey(Object key)`	returns `true` if key in map
• `containsValue(Object value)`	returns `true` if value in map
• `entrySet()`	returns a `Set` view of the key-pair mappings (`Set<Map.Entry<K,V>>`)
• `keySet()`	returns a `Set` view of the keys
• `values()`	returns a `Collection` view of the values

MAP<K, V>

Some special methods

- `replace(K key, V value)` => put if `containsKey(key)`
- `replace(K key, V oldVal, V newVal)` => put if `oldVal.equals(get(key))`
- `getOrDefault(K key, V default)` => get if `containsKey(key)`, otherwise `default`
- `putIfAbsent(K key, V value)` => put if `!containsKey(key)`

More advanced…

- `compute`, `computeIfAbsent`, `computeIfPresent`…

ADDITIONAL NOTES

- Always use `List<...>`, `Set<...>` as field / variable types, never `ArrayList`, `HashSet`, etc.
- Special **`for`** syntax for collections (iterables)

```
for (var item : collection) {
    System.out.println("Item " + item);
}
```

- `ConcurrentModificationException` when removing while iterating in a **for** loop (see video topics)
- Careful with `Lists` of numbers e.g. `List<Integer>` and the `remove` method:
  - `list.remove(1);` `// remove item 1 (2nd item)`
  - `list.remove((Integer) 1);` `// remove the first item equals to 1`

DATASTRUCTURES

Tools for structuring, organizing, managing data

Examples:

• Lists	To-do lists, playlist, high-score table, frames of an animation
• Stacks	Undo history, call stack in recursion, parsing
• Queues	Print job queue, task scheduling, game matchmaking
• Priority queues	Shortest path algorithm, CPU scheduling, calendar reminders, simulation engines
• Hash sets	Unique visitors, track seen states, prevent duplicate entries
• Hash tables	Mapping student ID to grade, words to frequencies, index, settings
• Trees	File system hierarchy, syntax trees, parse trees, XML/HTML, search trees in games, autocomplete/prefix trees
• Graphs	Social networks, routes on a map, software dependencies, recommendation systems (item similarity)
• Bitsets	Fast membership tests, bloom filters, tracking which seats are occupied

ALGORITHMS

Examples

• Searching	in a list: linear/binary, a graph: BFS/DFS, shortest path algorithms
• Divide and conquer	mergesort/quicksort, binary search
• Dynamic programming	knapsack problems, fibonacci with memoization
• String algorithms	parsing (with automata), tokenization, longest repeated substring
• Union-find	used to compute min-cut partition
• Backtracking	for example solving Sudoku, similar to search, constraint problem solving
• Randomized algorithms	Monte Carlo methods, randomized quicksort, hashing
• Compression	Huffman coding, LZW compression, run-length encoding
• Cryptography	hash functions, digital signatures, key exchange
• Machine learning	decision trees, k-means clustering, linear regression

Algorithms often require specific data structures and vice versa

COMPUTATIONAL COMPLEXITY

"Computational compexity" Big-O notation (Module 7)

• $O(1)$	constant time	accessing array, hash table lookup, swapping two variables
• $O(\log n)$	logarithmic time	binary search in a sorted array (cut problem in half each step)
• $O(n)$	linear time	array search, counting (check each element once)
• $O(n \log n)$	quasi-linear time	mergesort, fast fourier transform, balanced trees, divide-and-conquer
• $O(n^2)$, $O(n^3)$, …	polynomial (quadratic, cubic)	bubble sort (check all pairs, triples, etc, nested loops)
• $O(2^n)$	exponential	many optimisation problems

Does it matter if you choose a List or a Set?

COMPLEXITY OF COLLECTIONS

	LinkedList	ArrayList	HashSet
Insert	O(n)+O(1) (except head/tail)	O(n) (except tail)	O(1)
Remove	O(n)+O(1) (except head/tail)	O(n) (except tail)	O(1)
Search	O(n)	O(n) or O(log n) when sorted	O(1)
Iteration	O(n)	O(n)	O(n)
Index access:	O(n)	O(1)	N/A

So when to use what?

HOW TO DECIDE WHICH COLLECTION?

- Storing key-value pairs?
  - Yes: Use a `Map`: `HashMap`
  - No: Do you need an **order** (or **duplicate elements***)?
    - Yes: Use a `List`
      Fast random access: `ArrayList`
      Frequest random insertions/removals: `LinkedList`
    - Yes*: Can use a `Map<E, Integer>` to record duplication
    - No: Use a `Set`: `HashSet`
- To store associations, typically use a `HashSet`!

WHAT DATA STRUCTURE SHOULD WE USE?

- Ensure no duplicate usernames in a registration system
- Read some numbers and compute the median
- Manage a collection of products and their prices
- A way to quickly find products in a shop by their price
- Record a score for players in a tournament
- Record log entries for events
- Remember number plates we have seen before
- A shopping cart with items

OVERVIEW OF WEEK 4 EXERCISES

• 1	Checking code style
• 2	JML
• 3 – 4	Sorting
• 5	Linked lists
• 6 – 9	Tic Tac Toe
• 10 – 13	Map operations and JML
• 14 – 15	Stack calculator, unit testing

SUMMARY SO FAR

Java Collections

- List ArrayList, LinkedList
- Set HashSet
- Map HashMap

Any questions about **Java Collections**?