Conceptual Modeling and Programming in GIScience

Lecture 3: Methods and Object Orientation II

Yingjing Huang
yingjing.huang@univie.ac.at

2026S – April 27, 2026

Assignment Review

Public fields

```
public class Point {
    public double x, y;
    public Point(double x, double y) { ... }
}
```

Outside code can write `p.x = -999` with no check

```
Point p = new Point(3, 4); p.x = -999; // compiles fine
```

Private + getters

```
public class Point {
    private double x, y;
    public Point(double x, double y) { ... }
    public double getX() { return x; }
    public void setX(double x) { ... }
}
```

Outside code goes through your `getX/setX` gate

```
Point p = new Point(3, 4); p.x = -999; // compile error
```

- Public fields skip every check the class might want to enforce
- Private fields + getters make the class boundary real

Assignment Review

Week 2 Assignment

Reading: Read about methods and classes, and describe them in your own words (a short text is fine).

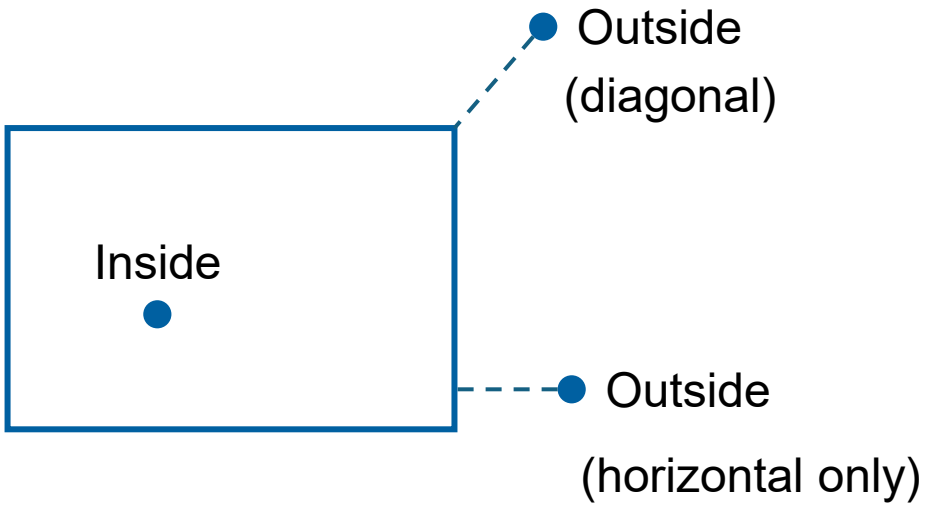
Coding:

- Define a `Point` class with a `distanceTo` method
- Define a `BoundingBox` class with an `isInside` method
- Write a test class that checks whether points fall inside the bounding box and computes distances between them

Submission: Upload `FirstNameLastNameW2.zip` with your `.java` files

Deadline: Sunday at 5:00 PM (Vienna Time)

Compute the distance of Point to BoundingBox



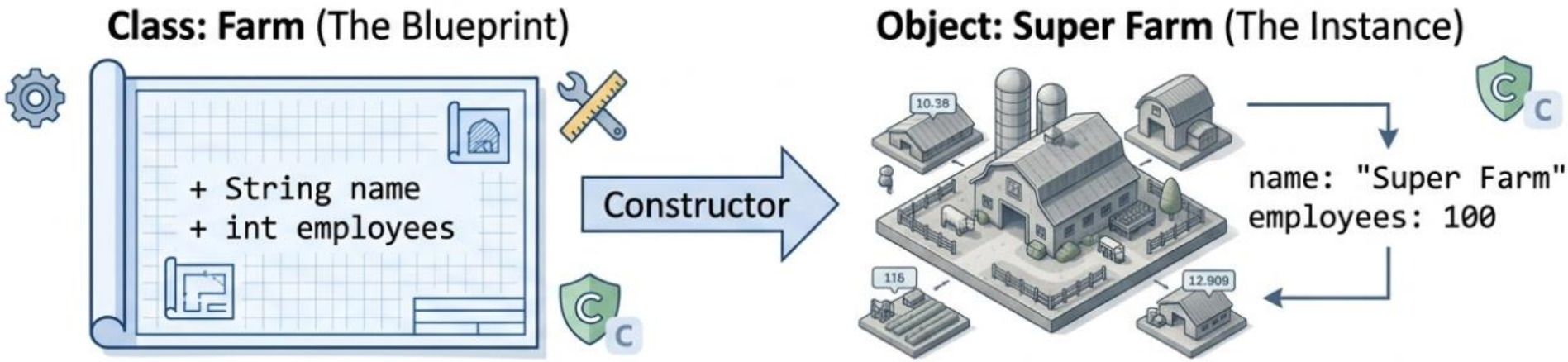
- If the point is inside: distance = 0
- If outside: shortest distance to the nearest edge

```
dx = max(xmin - px, 0, px - xmax)
dy = max(ymin - py, 0, py - ymax)
distance = sqrt(dx*dx + dy*dy)
```

What You Will Learn in This Lecture

- Review core concepts from last week
- Learn how to organize your classes with **packages**
- Learn how **OGC Simple Features** standardize geo-data in a vector format
- Get a first impression of **inheritance** in Java
- Clarify what the ***static*** keyword actually does
- See when ***final*** is the right way to lock something down
- Use ***enum*** to name a fixed set of states
- Meet your second data structure — **lists** (Java's **ArrayList**) — as a flexible alternative to arrays
- Outline the spatial aspects relevant for simulating an **epidemic**
- (All of this with a heavy hands-on focus, learning by example)

Week 2 Recap: Classes and Objects



- A **class** is a blueprint — which fields and methods any object will have
- An **object** is an instance built from the blueprint, with real field values
- A **constructor** runs when you create an object
- A class can have **multiple constructors** (different parameter lists)
- **Getters and setters** are controlled read/write

  — they can reject bad values

Week 2 Recap: Encapsulation, static, Visibility and Scope

Encapsulation

- Last week: the lake depth example
- Today: applied to inheritance design

static

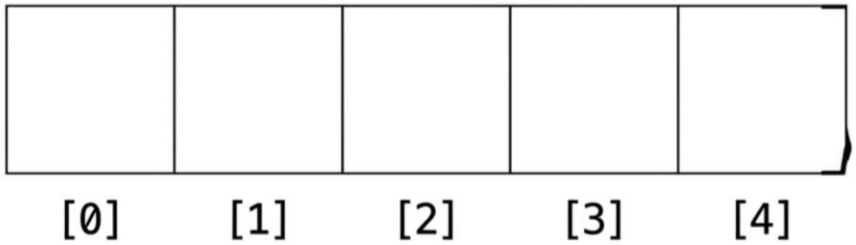
- Last week: belongs to the class, not to an object
- Today: three concrete uses

Visibility and scope

- Last week: public vs. private
- Today: protected joins, plus a fourth level from packages

Week 2 Recap: Arrays

Arrays provide **fixed capacity** collections accessed via **zero-based indexing**

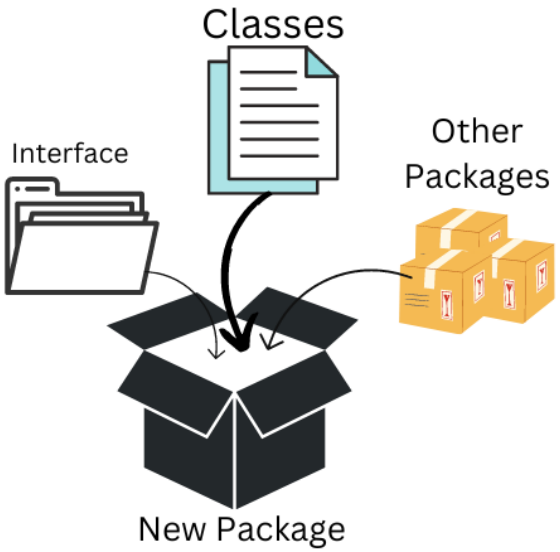



```
Point[] points = new Point[5];
```

Week 2 Recap: Pass-by-Value

- Primitive arguments pass **raw independent value copies** to methods
- Object arguments pass **reference copies** pointing to shared memory locations
- Modifications to referenced objects remain visible outside the method

- A package **groups related types** together
- It provides **access protection**
- It manages **namespaces**
- These types primarily include **classes and interfaces**



Code Example

Week1.java — Output, arithmetic, variables, loops, and methods.

```
package at.ac.univie.gis.week1;
```

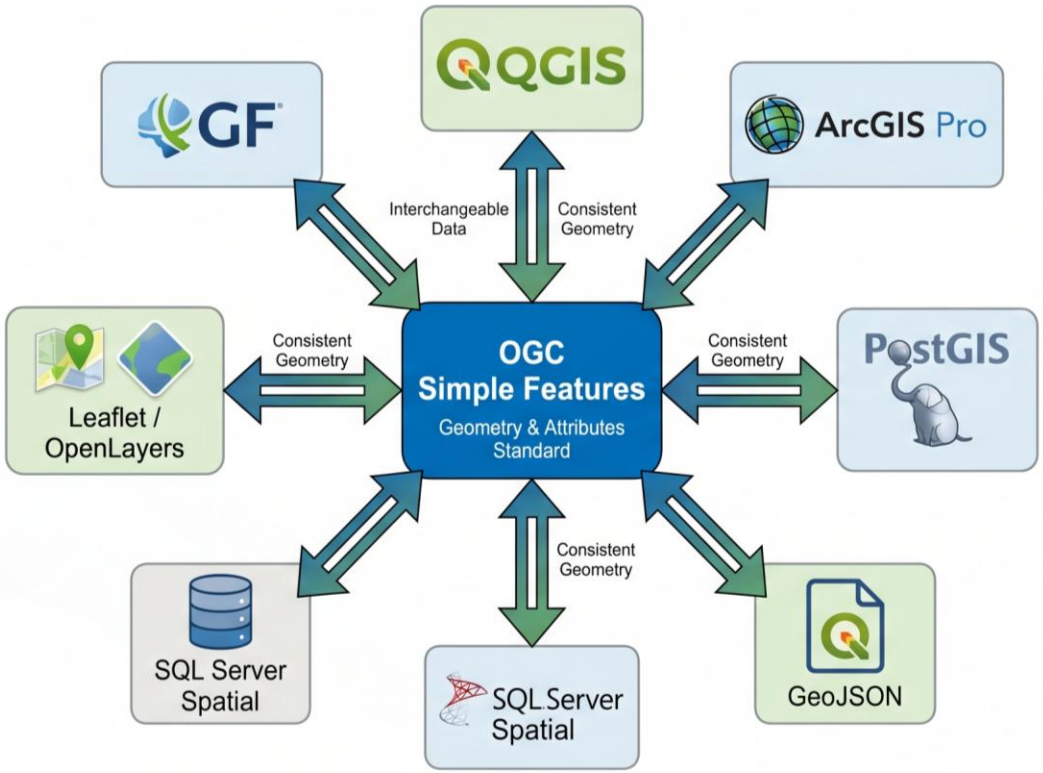
Packages Introduce a New Visibility Level

Access Modifiers

Modifier	Class	Package	Subclass	Global
Public	✓	✓	✓	✓
Protected	✓	✓	✓	✗
Default	✓	✓	✗	✗
Private	✓	✗	✗	✗

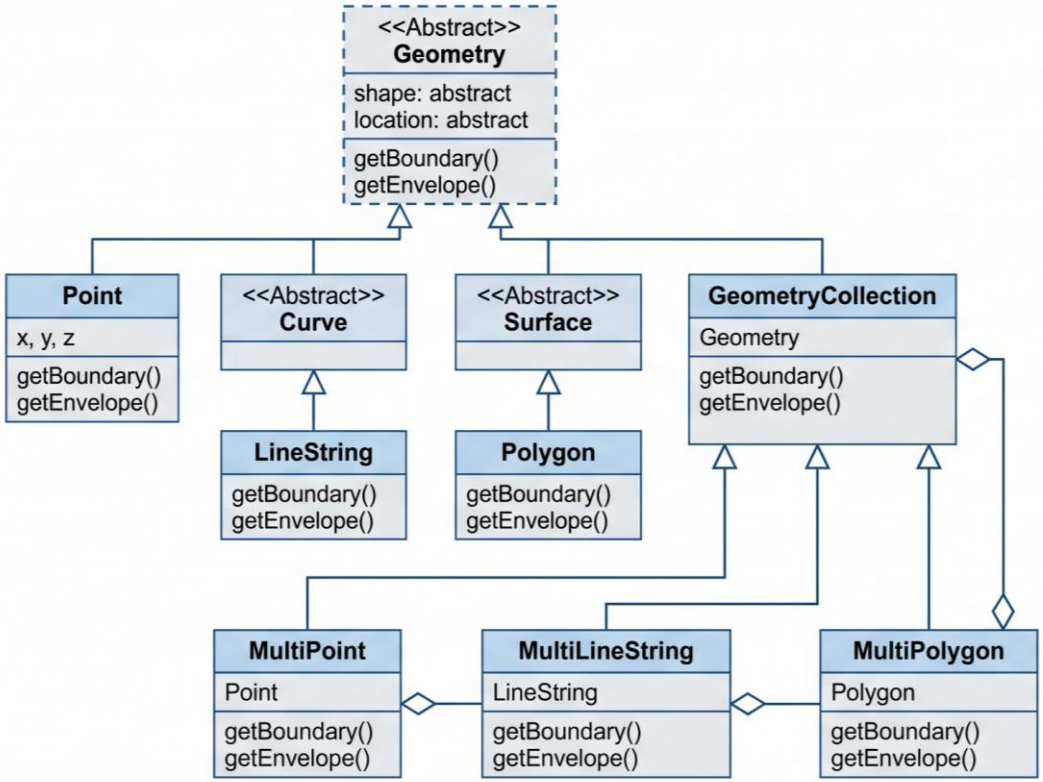
- Packages establish strict access boundaries beyond simple file organization
- Omitting a visibility keyword defaults to package private access

OGC Simple Features: Why Standards Matter



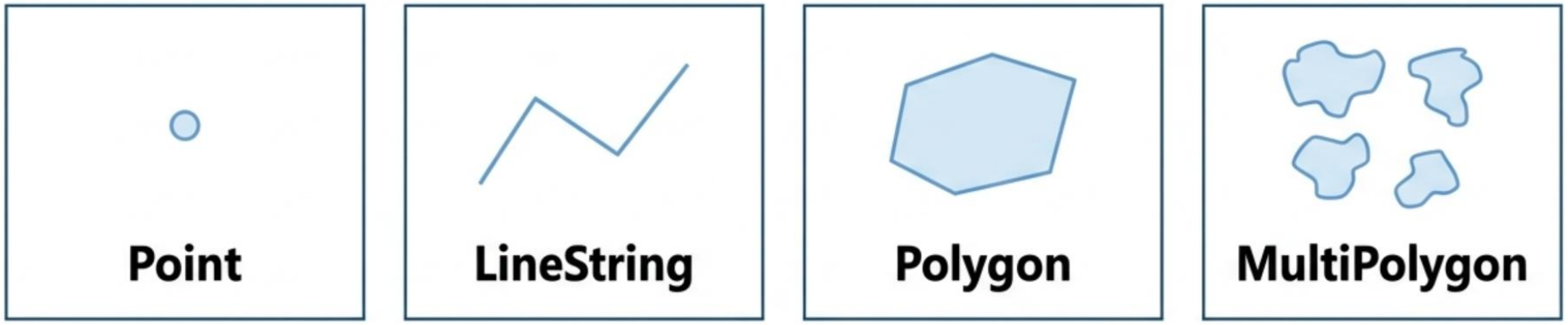
- Standardized geometry models ensure **cross-platform interoperability**
- Supported uniformly by **major GIS platforms and databases**
- Serves as an applied framework of **Object-Oriented Programming hierarchies**

OGC Simple Features: The Geometry Hierarchy



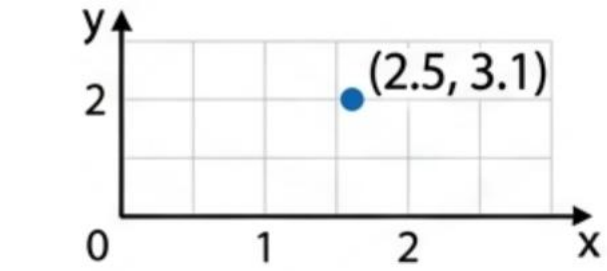
- The abstract **Geometry class** defines the foundational **shape and location** properties
- **Concrete implementations** include Point LineString and Polygon alongside their respective **Multi-collections**
- Structural relationships map directly to **Java inheritance**

OGC Simple Features: Concrete Examples

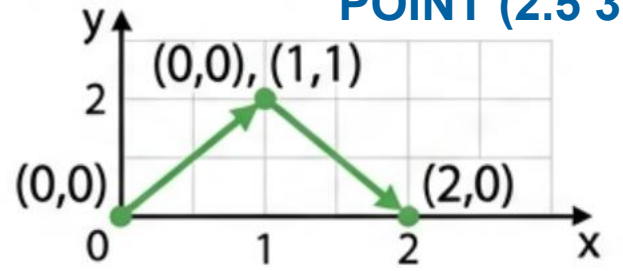


- A **Point** represents a single coordinate location
- A **LineString** consists of ordered points connected by straight segments
- A **Polygon** defines a closed area bounded by rings
- **Multi-geometries** group multiple separate shapes together

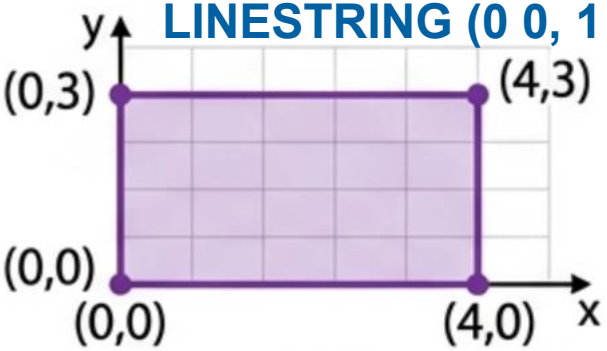
The abstract **Geometry** class defines shared operations including **bounding box area and intersection**



POINT (2.5 3.1)



LINESTRING (0 0, 1 1, 2 0)

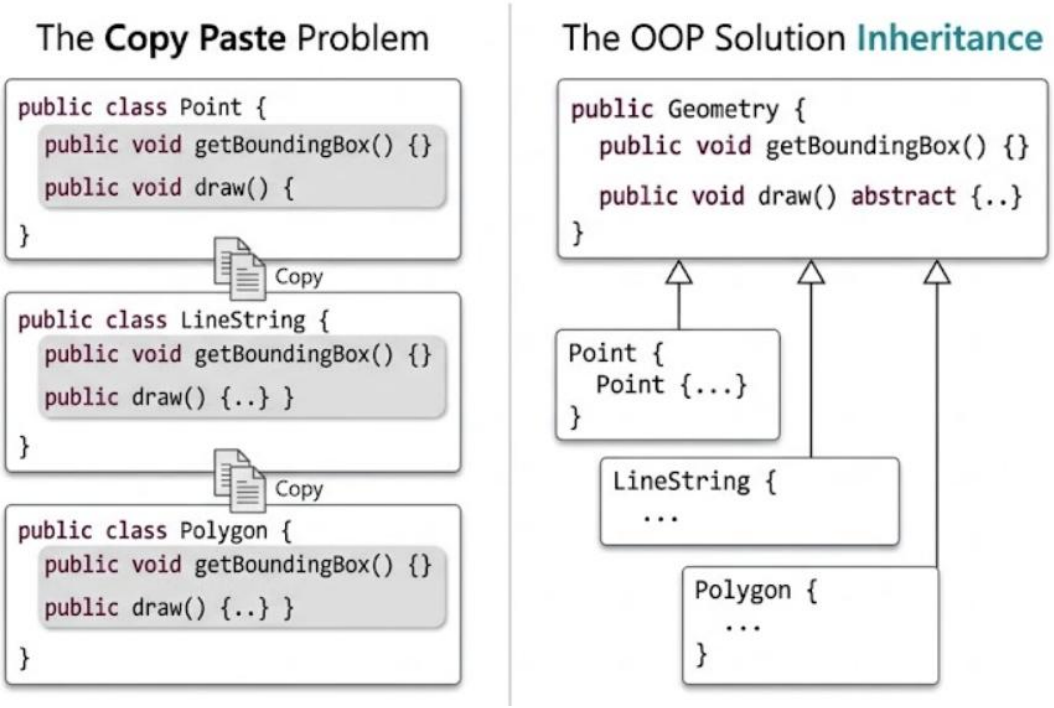


POLYGON ((0 0, 4 0, 4 3, 0 3, 0 0))

- Well-Known Text (WKT) serves as the **universal text serialization** for spatial geometries
- The string explicitly encodes the **geometry type and coordinate sequence**

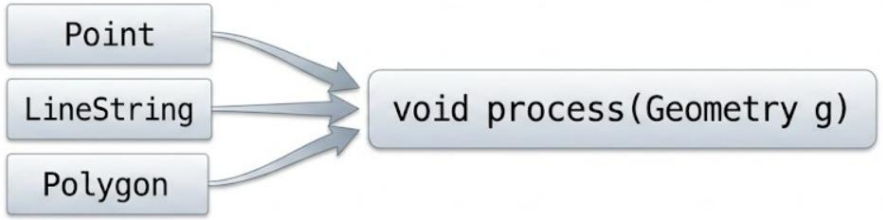
WKT defines geometry type and coordinate sequence, providing universal human-readable text serialization for spatial geometries and data exchange

The Copy-Paste Problem: Why We Need Inheritance



- **Duplicating** identical methods across multiple classes creates severe maintenance risks
- **Centralizing** shared logic into a single parent class prevents inconsistent behavior
- **Inheriting** capabilities allows child classes to receive all parent methods automatically
- **Establishing** a fundamental is a relationship helps the compiler understand your architecture

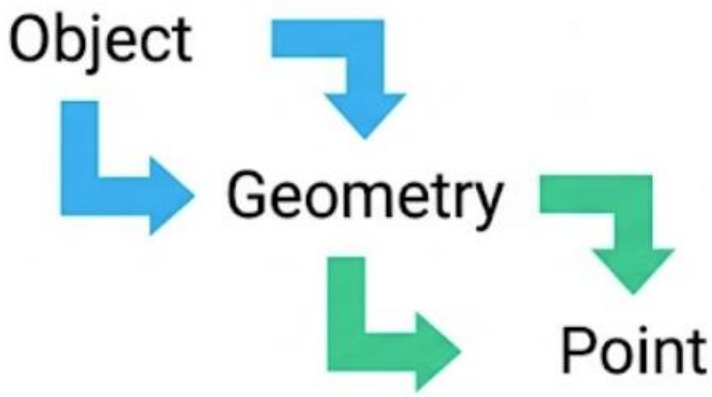
```
class Point extends Geometry {
  ...
}
```



A **Point** establishes an **is-a** relationship functioning anywhere a **Geometry** is expected

What's Inherited, What's Not

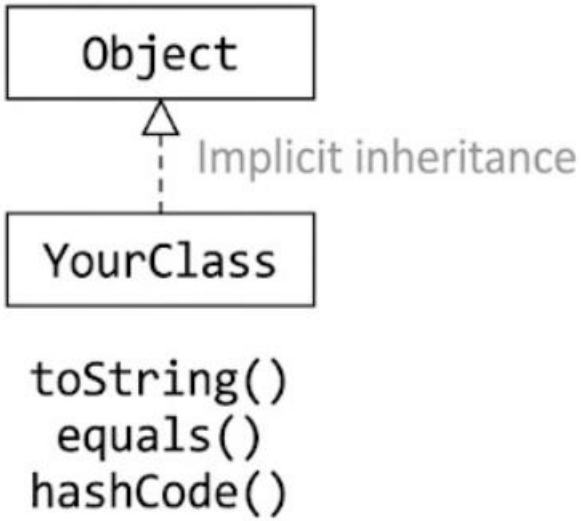
Feature	Inherited
Fields	✓
Methods	✓
Constructors	✗



Parent construction runs first top down

- **Fields** and **methods** transfer to the child class **automatically**
- **Constructors** remain permanently exclusive to their defining class
- Java **systematically executes parent constructors** before child code runs
- Object creation always flows top down from the highest parent class

Every Class Has a Silent Parent: Object

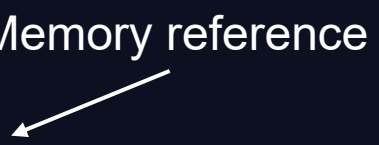


- Every Java class silently extends the ***Object*** Class
- The Object class servers **as the universal ancestor** providing foundational method
- The default toString method returns a memory address which is rarely useful
- You will typically override these default implementations in your own classes

Method Overriding: toString() as a Case Study

- Every Java class implicitly inherits from the universal **Object class**
- All objects possess a built-in **toString method** by default
- The default implementation returns the class name and a **memory reference**

```
package at.ac.univie.gis.week3;

public class PointTest {
  public static void main(Sting[] args) {
    Point p1 = new Point(10,10);
    System.out.println(p1);
  }
}

// Output: at.ac.univie.gis.week3.Point@527c6768
```



Overriding toString(): Example

- **Overriding** allows developers to replace inherited methods with custom implementations
- The new method **signature must remain identical** to the parent method
- The custom string provides **readable and meaningful output** for debugging

```java
package at.ac.univie.gis.week3;

public class Point {
    private double x;
    private double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public String toString() {
        return "(" + x + ", " + y + ")";
    }
}
```

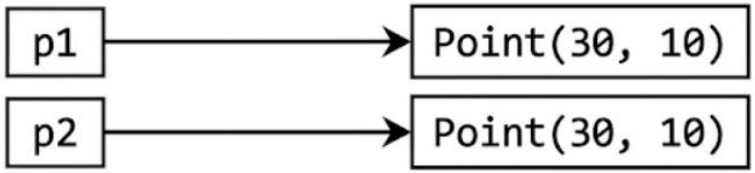
```java
package at.ac.univie.gis.week3;

public class PointTest {
  public static void main(Sting[] args) {
    Point p1 = new Point(10,10);
    System.out.println(p1);
  }
}

// Output: (10.0,10.0)
```

Overloading vs Overriding: Don't Mix Them Up

Feature	Overloading	Overriding
Decided when	compile time	runtime
Relationship	Single class	Parent-Child (inheritance)
Method Signature	Different parameter list, same method name	Identical method name and parameter list
Resolved By	Compiler	Runtime Environment

- Overloading occurs within a single class using different parameter lists
- Overriding occurs across an inheritance boundary using the exact same signature
- The compiler resolves overloading while the runtime environment resolves overriding

equals vs ==: Identity vs Equality



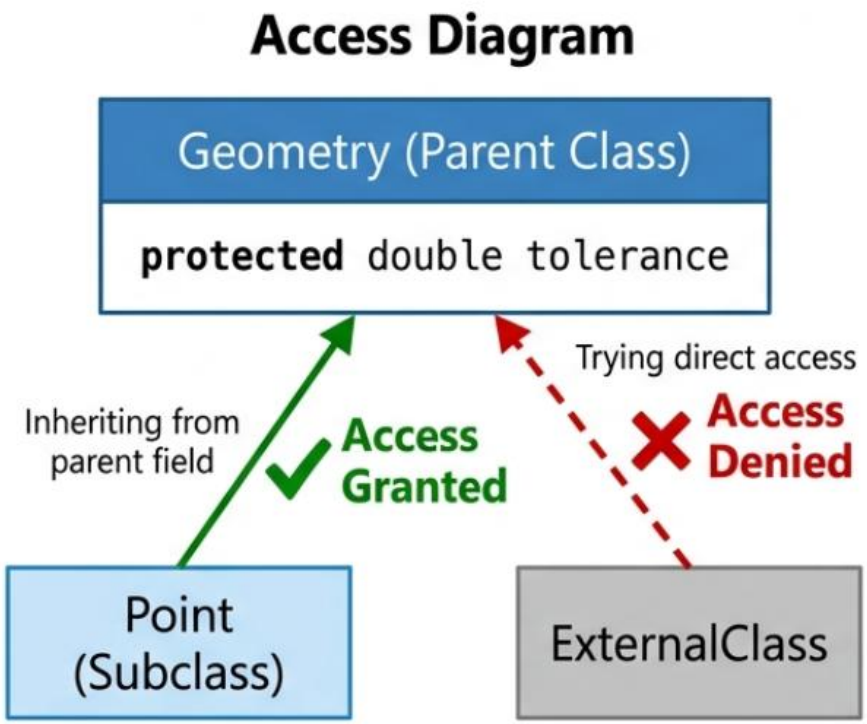

```
p1 == p2; // false
p1.equals(p2); // false
```

```
@Override
public boolean equals(Object obj) {
    if (this == obj) return true;

    if (obj == null || getClass() != obj.getClass()) return false;

    Point other = (Point) obj;
    return this.x == other.x && this.y == other.y;
}
```

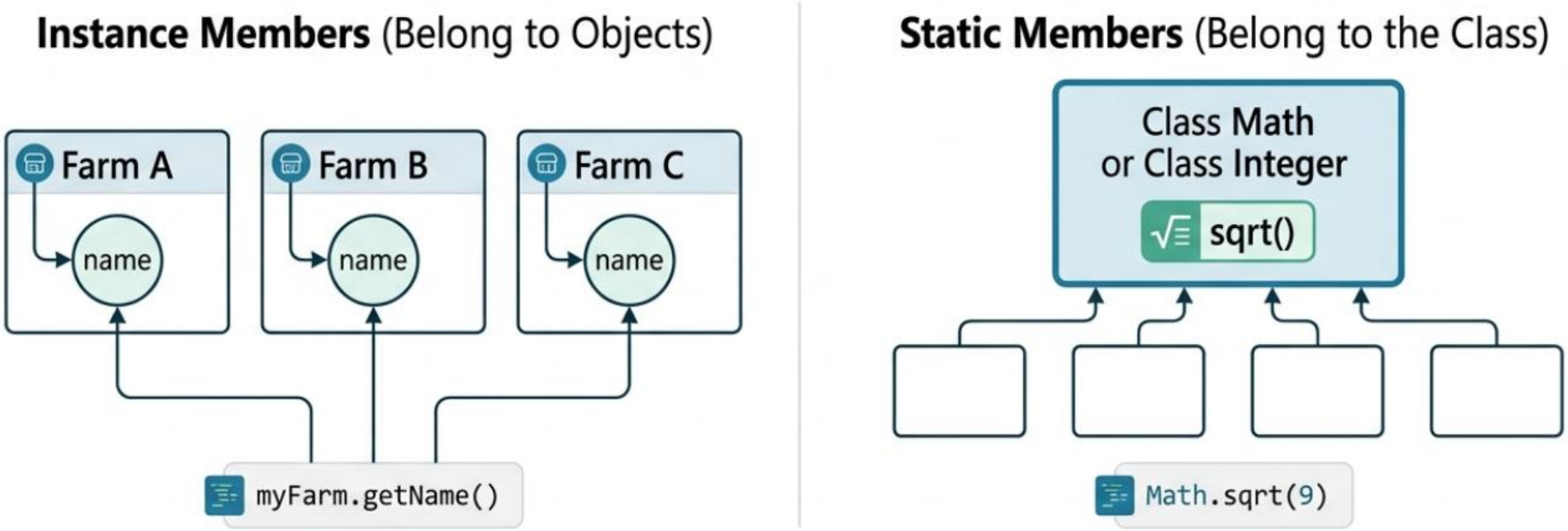
```
// Override equals

p1 == p2; // false
p1.equals(p2); // true
```

- Comparing **primitives** relies safely on ==
- Evaluating objects with == checks only their memory address
- Calling the **default equals** method also checks only the memory address
- **Overriding** the equals method guarantees true value-based comparison

Revisiting protected with Inheritance



- The **protected modifier** grants direct access to inheriting subclasses
- It serves as an architectural middle ground between **private and public visibility**
- Tight coupling to parent fields can cause **subclass breakage** during updates
- Best practice favors exposing **protected methods** rather than raw fields to ensure safety



- **Instance members** belong to individual objects and require instantiation
- **Static members** belong to the class itself and are **shared globally**
- Static methods are invoked directly via the **class name**
- The main method is static because it executes **before any objects exist**

Constants

```
public static final double EARTH_RADIUS_KM = 6371.0;
```

Constants establish global fixed values that never change

Utility Methods

```
public static double distanceBetween(Point a, Point b) {
  /* method logic */
  {...}
}
```

Utility methods perform stateless helper functions without requiring object instantiation

Shared Variables

```
private static int totalPersonsCreated = 0;
```

Shared variables maintain common data and counters across all class instances

final: When You Want Things to Stop Changing

```java
// Variable case

final int level = 1;
level = 2;  // Compiler error: cannot assign a
value to final variable

// Method case

class Base {
    public final void draw() {
        // Core logic
    }
}

class Derived extends Base {
    @Override
    public void draw() {
        // Compiler error: overridden method
is final
    }
}

// Class case

public final class SystemConfig {
    // Configuration logic
}

class MyConfig extends SystemConfig {
    // Compiler error: cannot inherit from
final class
}
```

- Locking variables strictly prevents any reassignment after initialization
- Securing methods prohibits subclasses from overriding critical internal logic
- Sealing classes completely forbids any architectural inheritance

Modeling case: Traffic light system

String

```
light.setState("red");
```

Risk:

- Prone to typos
- “RED” is different from “red”
- Errors only found at runtime
- No compile time validation

int Constant

```
light.setState(0);
```

Risk:

- What does 0 mean?
- Hard to read and maintain
- Easy to assign wrong values
- No compile time validation

enum

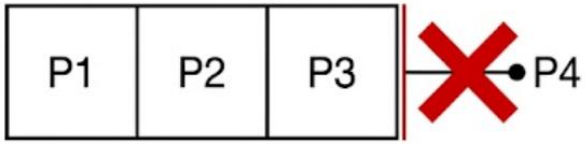
```
enum TrafficLight {
  RED,
  YELLOW,
  GREEN
}
light.setState(TrafficLgiht.RED);
```

Benefits:

- Strict **fixed set of name values**
- Compile time validation
- Completely prevents typos
- Absolute **type safety**
- Powerful **autocomplete** support

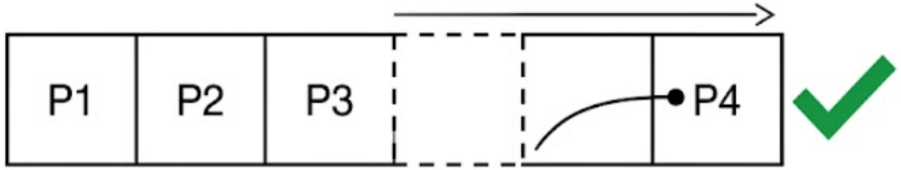
The Array

Fixed Size



The ArrayList

Dynamic Growth



- Standard arrays possess **fixed lengths** determined at creation time
- The ArrayList data structure provides **dynamic resizing** to accommodate new elements automatically
- Angle brackets enforce **strict type safety** indicating exactly what objects the list can hold
- Developers must include an explicit **import statement** from the java.util package

```
import java.util.ArrayList;
ArrayList<Point> myPoints = new ArrayList<>();
```

The Cheat Sheet

Core Methods

Method	Action
`add(e)`	append element
`add(i, e)`	insert element at index
`get(i)`	retrieve element
`set(i, e)`	modify element at index
`remove(e)`	delete element
`remove(i)`	delete element at index
`size()`	check list size

Iteration

Iteration supports both classic index loops and **enhanced for loops**

```
for (Point p : pointList) {
  // Perform operations with each Point p
}
```

Think about

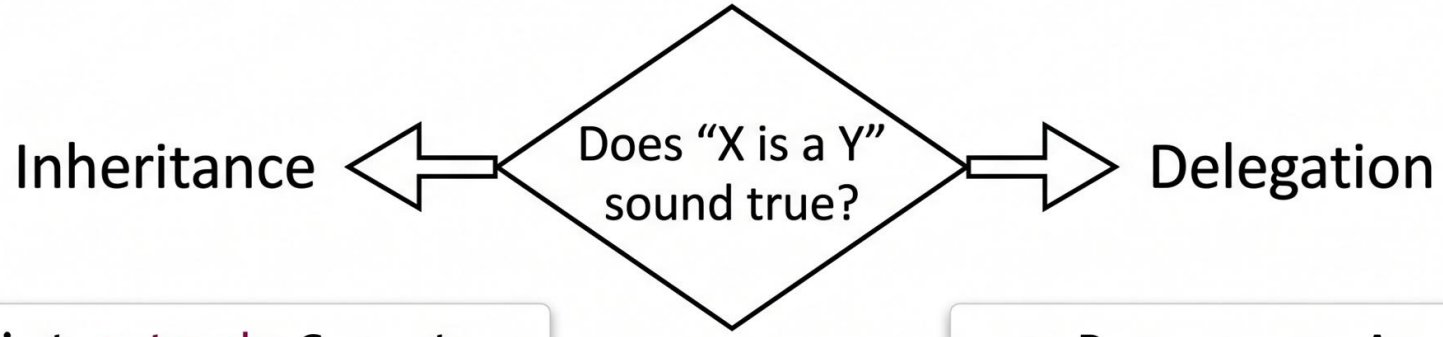
How would you use a ‘for’ loop to draw a ‘PolyLine’ from these points?

Delegate to ArrayList in PolyLine

```java
public class PolyLine {
    private ArrayList<Point> points = new ArrayList<>();

    public void addPoint(Point p) {
        points.add(p); // Delegation in action
    }
}
```

- The PolyLine class maintains an internal ArrayList to manage **data storage**
- The custom addPoint method **delegates** the actual operation to the underlying list
- This structural pattern demonstrates behavior **composition** rather than code duplication
- Delegation represents a **has-a relationship** while inheritance defines an **is-a relationship**

Inheritance vs Delegation: Which One?



e.g., Point extends Geometry

```
class Point extends Geometry { ... }
    ...
```

A specialized Geometry

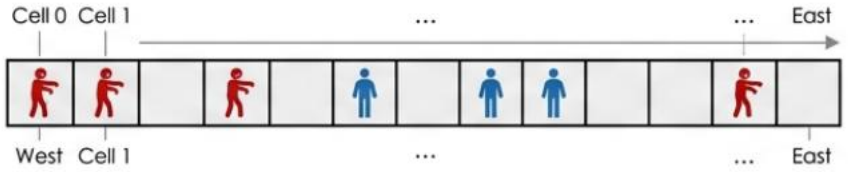
e.g., Person uses ArrayList

```
class Person {
    ArrayList<Point> points;
    ...
}
```

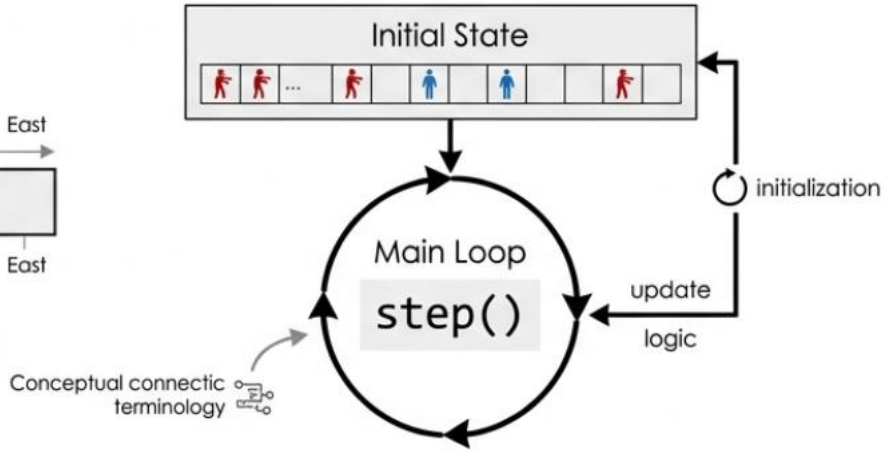
Tight utilizes Point

When in doubt **delegation**

The Playing Field



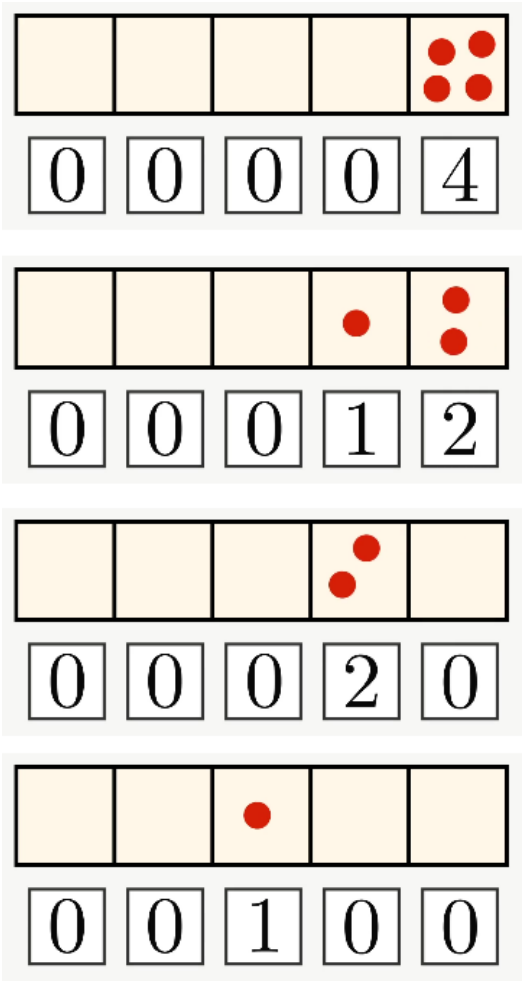
The Simulation Architecture



- The simulation requires a structured **playing field** using a coordinate grid
- An **initial state** defines the starting placement of all entities
- The **step method** advances the simulation logic from one discrete state to the next
- A continuous **main loop** drives the execution until a termination condition is met

The Interaction Rule

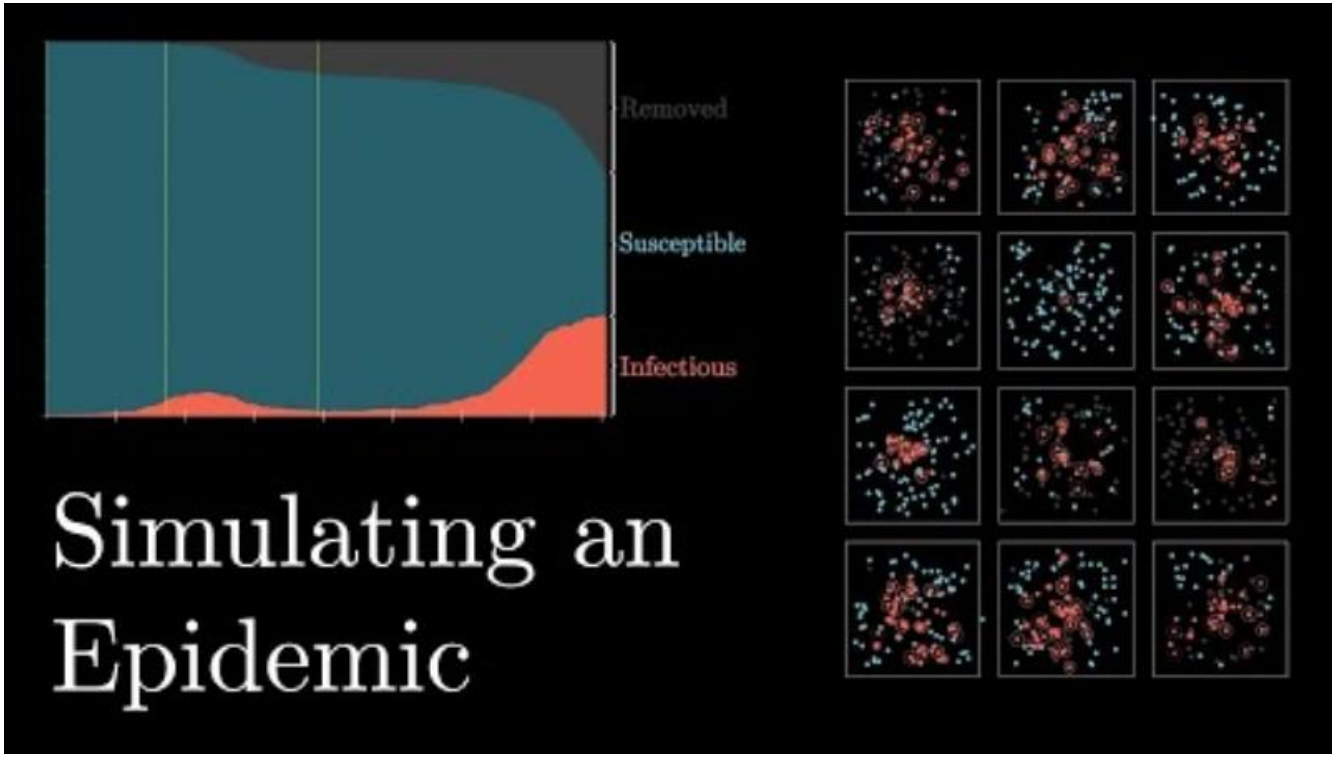
- If a cell contains multiple zombies one moves left and one disappears
- The simulation repeats until all cells contain one or zero zombies



Challenge

- What data structure optimally represents a linear corridor?
- How do we handle edge cases at index zero?

Simulating an Epidemic



- Check out the video at

  https://www.youtube.com/watch?v=gxAaO2rsdIs

- The video does **not** offer any medical advice and neither do we.
- Models are based on abstraction an generalization.
- Let us create (parts of) a very simple SIR simulation together.

Reading

- Read chapters in your book about the keyword static, the ArrayList, get familiar with the Java API, e.g., Arrays class.

Coding

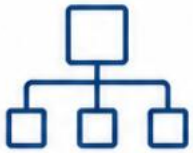
- Create a class for people representing **susceptible**, **infectious** or **recovered** states
- Assign randomly generated static locations to each person without continuous movement
- Define a rectangular study area and select people located inside it
- Evaluate infection risk using specific distance radius and probability chance
- Validate your internal logic but **do not try to run a continuous simulation** yet

Submission

- Upload a zip file **[FirstNameLastNameW3.zip]** with the *.java files
- **Deadline**: Sundays at 5:00 PM (Vienna Time) (The day before our Monday lectures).

Code Example

inheritance/



Geometry parent class and overriding

polyline/

ArrayList delegation pattern

zombies/

Complete corridor simulation loop

Check code at **GitHub** or **Moodle**