Conceptual Modeling and Programming in GIScience

Lecture 4: Inheritance and Polymorphism I

Yingjing Huang
yingjing.huang@univie.ac.at

2026S – May 4, 2026

Week 3 Assignment

- 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

What You Will Learn in This Lecture

- Chain constructors with ***super*** and ***this***
- Understand **dynamic dispatch** during method overriding
- Handle mixed-type collections safely with ***instanceof*** and **downcasting**
- Declare **abstract class** to enforce implementation contracts
- Formalize the concept of **Polymorphism**
- Avoid classic inheritance pitfalls like tight coupling
- Add controlled randomness with ***java.util.Random***
- Extend the SIR simulation with **probability-driven infection**

(We will discuss all this by extending last week's SIR simulation)

extends + is-a

Point can be used wherever a *Geometry* is expected

```
public class Point extends Geometry {
    // Point IS-A Geometry
}
```

Overriding (*@Override*)

Child replaces the parent's version with the same signature

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

```java
public Farm(String name) {
    this.name = name;
}

public Farm(String name, int nrOfEmployees)
{
    this.name = name;
    this.nrOfEmployees = nrOfEmployees;
}
```

Overloading

Same name, different parameters.

Compiler picks the version based on the arguments at compile-time.

Constructors are NOT inherited

- Each class writes its own constructors
- Java auto-inserts *super()* when the parent has a no-arg constructor

HealthState *enum*

```
public enum HealthState {
    SUSCEPTIBLE,
    INFECTIOUS,
    RECOVERED
}

public class Person {
    private HealthState state;
}
```

- Compiler-checked white-list of named states
- Switch on it cleanly, no integer or string magic values

static ID counter

```
public class Person {
    private static int nextId = 0;
    private final int id;

    public Person() {
        this.id = nextId++;
    }
}
```

- Shared across all Person instances — one counter, the class owns it
- Every new Person gets a unique, monotonically increasing ID

ArrayList<Person>

```
import java.util.ArrayList;

ArrayList<Person> population = new ArrayList<>();
population.add(new Person());
population.add(new Person());

for (Person p : population) {
    System.out.println(p);
}
```

- Dynamically resizable — no fixed-size limit at creation
- Generic type <Person> makes it compiler-checked

---

public static final constants

```
public class Disease {
    public static final double INFECTION_PROBABILITY = 0.3;
    public static final double INFECTION_RADIUS      = 5.0;
}

// usage:
if (random.nextDouble() < Disease.INFECTION_PROBABILITY) { ... }
```

- *static* — class-level, one shared copy in memory
- *final* — assigned once, never reassigned
- Reads naturally

How a Subclass Hands Data to Its Parent

Watercourse (parent)

```
public class Watercourse {
    private double length;            // private — owned by parent

    public Watercourse(int length) {
        if (length < 0)
            throw new IllegalArgumentException();
        this.length = length;         // validation + assignment
    }
}
```

River (child)

```
public class River extends Watercourse {
    private Spring origin;

    public River(int length, Spring origin) {
        // ❓ how does length get set?
        // this.length = length;        // compile error — private
        this.origin = origin;
    }
}
```

- Watercourse owns *length* and validates it in its own constructor
- ***length* is *private*** — River can't write to it directly
- River needs a way to ask Watercourse to do the setup itself

Writing super(...) in a Constructor

```
public class River extends Watercourse {
    private Spring origin;

    public River(int length, Spring origin) {
        super(length);              // hand length up to Watercourse
        this.origin = origin;       // then do River's own work
    }
}
```

- First statement only — no other line may come before *super*
- Called once per constructor — twice means something is confused

Mental model: parent first, child after.

What Auto-Inserted super() Actually Does — Overview

Three cases of what Java's auto-inserted ***super()*** actually does:

✅ **Case 1 — Success**

Parent has a no-arg constructor; child needs nothing from parent

⚠️ **Case 2 — Silent Trap**

Compiles, but data is lost because *super()* is called instead of *super(data)*.

❌ **Case 3 — Failure**

Parent has no no-arg constructor. Compiler rejects the code.

Case 1: Auto-Call Success

Setup: parent has a **no-arg** constructor; child needs **nothing** from parent

```
class Animal {
    public Animal() { ... }            // no-arg ctor exists
}

class Dog extends Animal {
    public Dog(String name) {
        // Java silently inserts: super();
        this.name = name;
    }
}
```

Outcome: Animal's no-arg constructor runs, then Dog adds its name

Fix: none needed

Case 2: The Silent Bug

Setup: parent has both no-arg and args versions; child wants to pass data up but forgets to write super

```
class Watercourse {
    public Watercourse() { length = 0; }        // no-arg
    public Watercourse(int length) {            // args version
        this.length = length;
    }
}

class River extends Watercourse {
    public River(int length, Spring origin) {
        // Java silently inserts: super();   ← calls Watercourse() — NOT Watercourse(int)
        this.origin = origin;
    }
}
```

Outcome: Compiles and runs. But the parent's *length* stays at 0 — the value you passed in was silently dropped

Fix:

```
public River(int length, Spring origin) {
    super(length);          // hand the length up explicitly
    this.origin = origin;
}
```

Case 3: Compiler Rejection

Setup: parent has only an args version, no no-arg

```
class Watercourse {
    public Watercourse(int length) { ... }     // ONLY this – no no-arg
}

class River extends Watercourse {
    public River() {
        // Java tries: super();   ← Watercourse() doesn't exist
    }
}
```

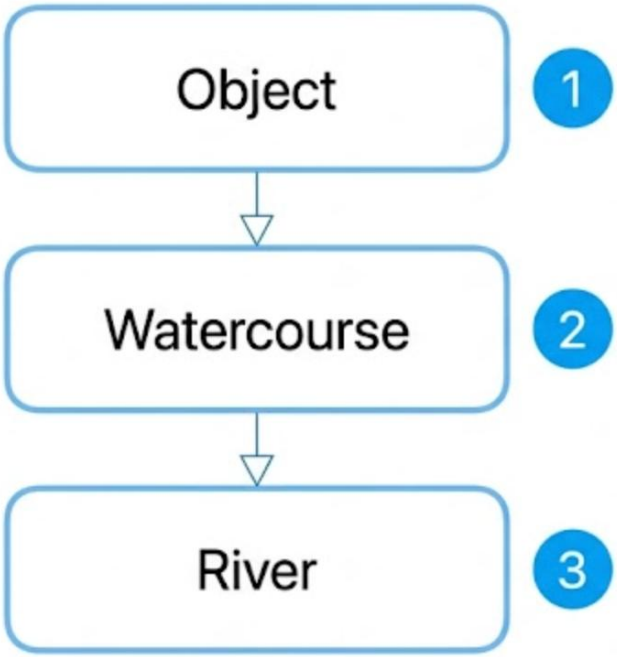
Outcome: Compile error

constructor Watercourse in class Watercourse cannot be applied to given types

Fix:

```
public River(int length) {
    super(length);          // call the args version with a real value
}
```

Construction Order: Top to Bottom



Construction order for *new River(120, springX)*:

1. **Object** constructor runs first — the silent universal parent
2. **Watercourse** constructor runs next — the direct parent (sets ***length***)
3. **River** constructor runs last — the child (sets ***origin***)

- Each level finishes before the next level starts
- By the time River's body runs, the parent side of the object is fully initialized
- This is exactly why *super(...)* must be the first statement — Java needs a guarantee that parent setup happens before child setup touches anything

Reusing a Constructor with *this(...)*

```
public class Rectangle {
    private double width;
    private double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    public Rectangle(double side) {
        this(side, side);          // delegate to the main constructor
    }
}
```

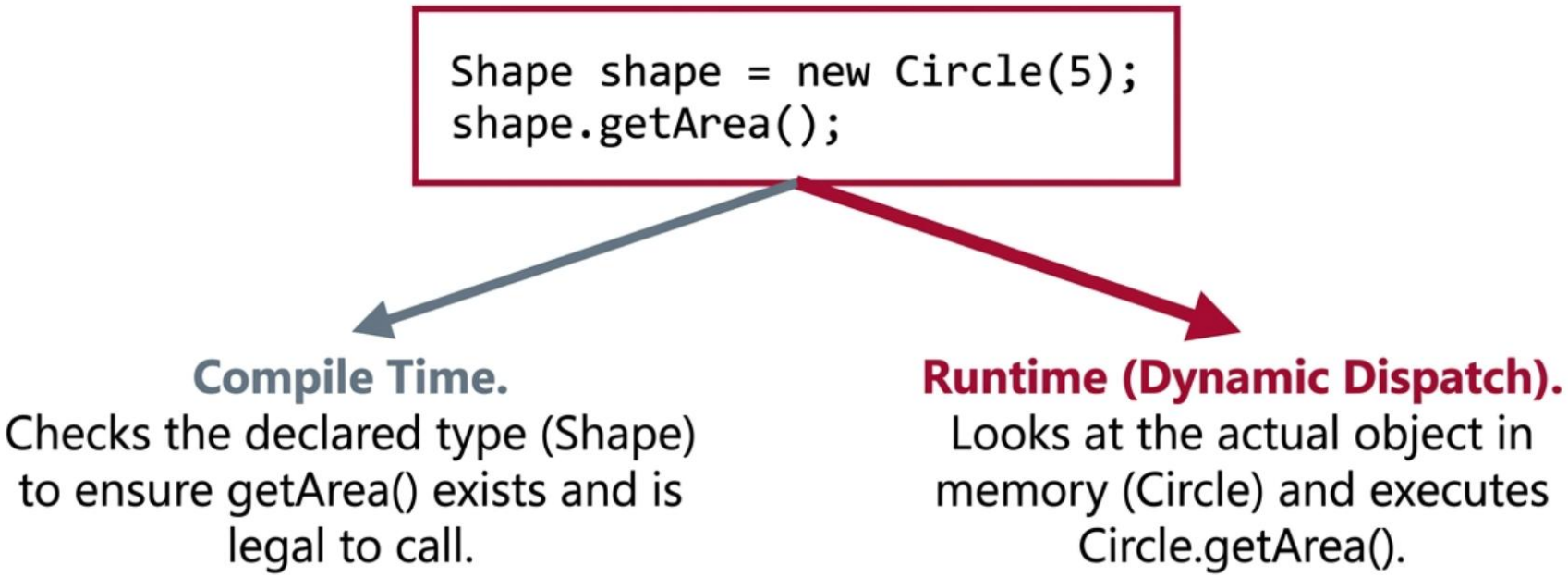
this(…) delegates within the same class

Rules (same as *super*)

- Must be the first statement in the constructor.
- Can only appear once per constructor.

The big rule

- A constructor starts with *super(...)* **OR** *this(...)*, never both.
- If you call *this*, the chained constructor eventually calls *super* for you.



What dynamic dispatch is

When you call a method on a variable, Java picks the version to run **at runtime, based on the actual object** — not at compile time, based on the variable's declared type.

Also called **late binding**.

Rules the Compiler Enforces on Overrides

Same name + parameters

double getArea() matches *double getArea()*

Return type same or covariant

Shape parent → *Circle* child OK

Visibility cannot narrow

public parent → must stay *public*

@Override catches typos

@Override double getArae()

@Override is non-negotiable.

Adding to the Parent with super.method()

```
public class PaintedCircle extends Circle {
    private String color;

    @Override
    public String toString() {
        return super.toString() + " color=" + color;
    }
}
```

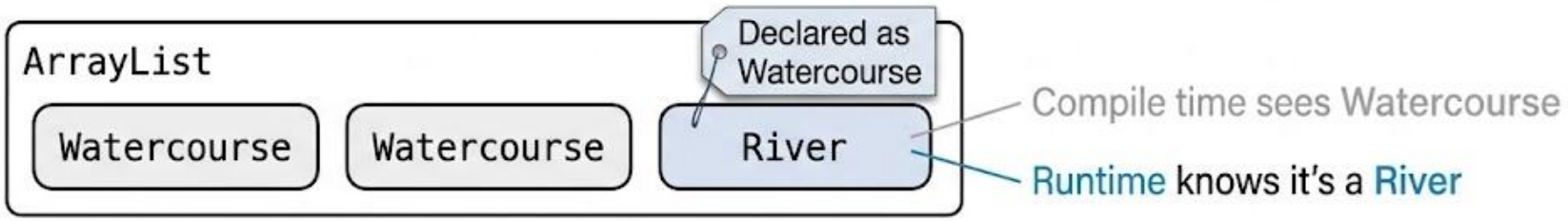
When to use

- The child's method should do everything the parent's does, **plus something extra**
- A classic use is *toString()* — the child's version reuses the parent's description and appends its own fields

Rules

- Can be called anywhere in the child's method body
- Can be called any number of times
- Can be called on any inherited method, not just the one being overridden

When the Compile-Time Type Hides the Real Type




```
ArrayList<Watercourse> list = ...;   // declared as Watercourse

for (Watercourse w : list) {
    w.getLength();      // OK – every Watercourse has length
    w.getOrigin();      // ❌ compile error – Watercourse has no getOrigin
}
```

- **Compile-time type** of each element: *Watercourse* (what the variable says it is)
- **Runtime type** of each element: could be *Watercourse* or *River* (what the object really is)
- The compiler only allows methods defined on the declared type — it has no way of knowing some elements are actually *Rivers*
- To safely access River-only methods, we need a runtime check — that's what *instanceof* is for

What *instanceof* does

Checks whether an object is of a given type — or a subtype of it.

Returns *true* or *false*.

Dangerous cast

```
River r = (River) item;      // assumes item is a River
r.getOrigin();               // ClassCastException at runtime if it isn't
```

Safe cast

```
if (item instanceof River) {
    River r = (River) item;
    r.getOrigin();           // safe, the check guarantees the cast succeeds
}
```

The rule: always guard a downcast with *instanceof* first. This check-then-cast pattern shows up in any Java codebase that works with heterogeneous collections

Pattern-Matching *instanceof* in One Line

Classic two-step (any Java version)

```
if (item instanceof River) {
    River r = (River) item;
    System.out.println(r.getOrigin());
}
```

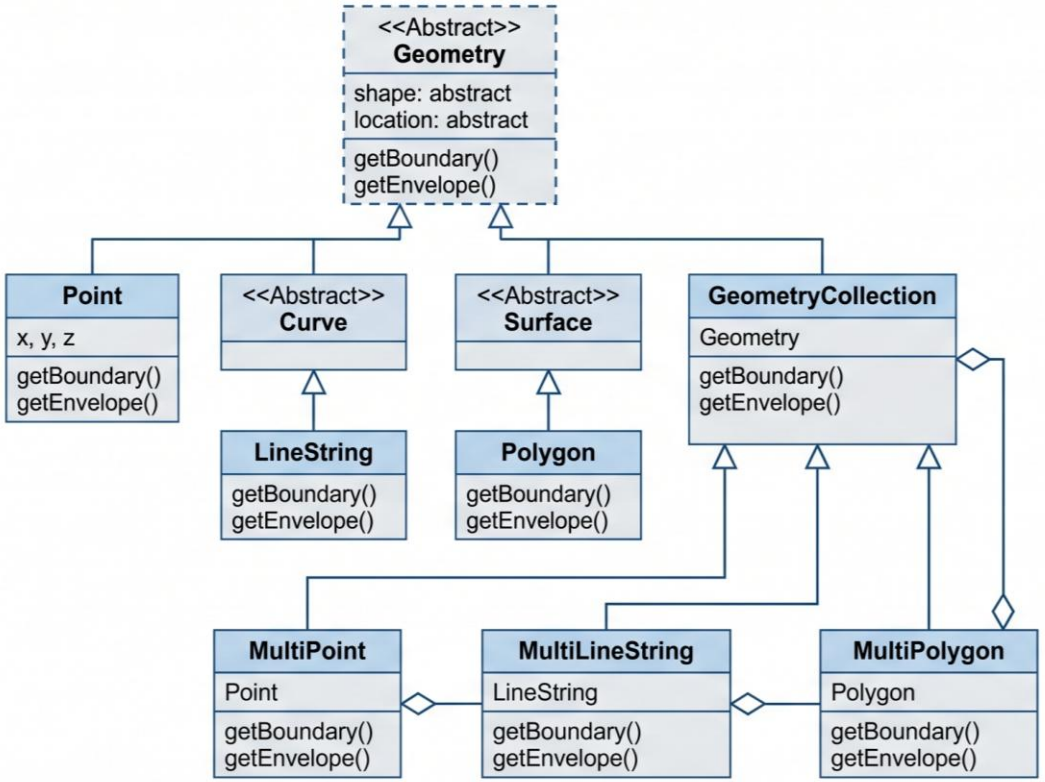
Since Java 16, the check and the cast can be combined into a single expression

Pattern matching (Java 16+)

```
if (item instanceof River r) {
    System.out.println(r.getOrigin());
}
```

The *instanceof* check declares and binds the cast variable inline

When a Parent Should Refuse to Be Built



Every subclass **must** provide its own *getArea()*

— so callers can rely on it

Nobody should be able to instantiate a bare *Geometry*

— *new Geometry()* is a modeling error

```
new Geometry()       // ❌ should be REJECTED
new Circle(5)        // ✅ OK — area = π × 25
new Rectangle(3, 4)  // ✅ OK — area = 12
```

Declaring abstract Classes and Methods

```
public abstract class Geometry {
    private String name;

    public abstract double getArea();    // no body – subclass fills in

    public String getName() {            // concrete shared method
        return name;
    }
}
```

- **abstract class**

  → no new

- **abstract method**

  → must override (or the subclass is also abstract)

Abstract Geometry Forcing the *getArea* Contract

- Every concrete subclass is **forced** to provide its own *getArea*
  — The compiler refuses to accept a subclass that doesn't.
- *new Geometry()* is rejected
  — no shapeless geometries can exist in your program.
- Shared methods can still live on *Geometry*
  — only the truly shape-specific parts stay abstract.

Calling code can treat any *Geometry* uniformly, **without knowing or caring which concrete shape it has**:

```
double total = 0;
for (Geometry g : list) {
    total += g.getArea();   // dispatches to each shape's own implementation
}
```

Choosing Abstract or Concrete

When to declare a class *abstract* — both must be true:

1. The parent represents a **general category that needs subtypes to be meaningful**
2. You **actively want to forbid *new* on the parent**

Examples:

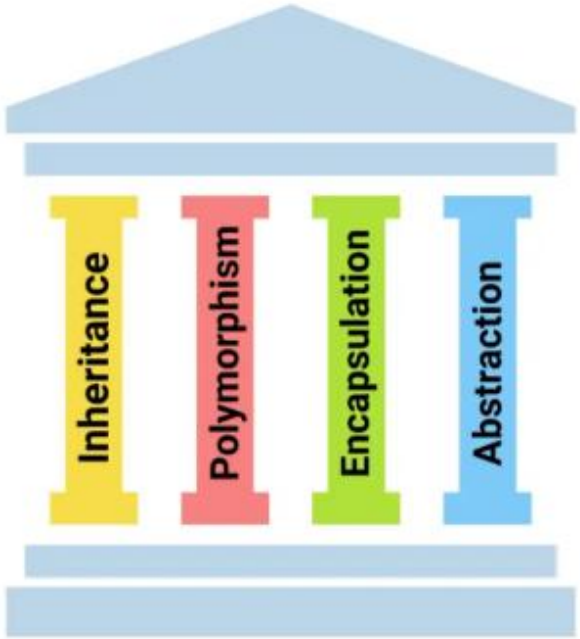
- *Geometry* → abstract
  (no meaningful default; we don't want `new Geometry()`)
- *Animal* → often concrete
  (an unspecified Animal can still have a name and an age)

Polymorphism

Writing code against a parent type that works transparently for any subclass

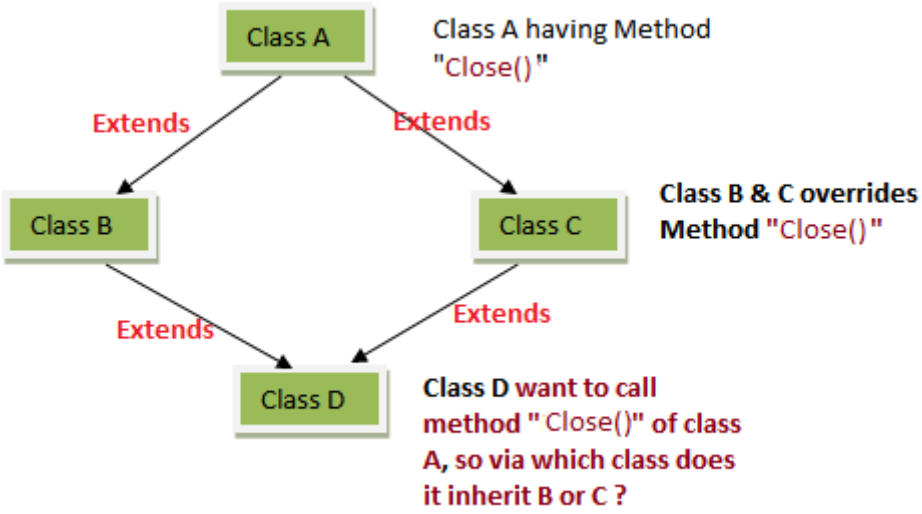
```
ArrayList<Watercourse> list = ...;

for (Watercourse w : list) {
    System.out.println(w.getLength());
    if (w instanceof River r) {
        System.out.println(r.getOrigin());
    }
}
```



Pitfall One: Java's **Single Inheritance**

Diamond Problem



- **Java rule:** a class can only *extends* one parent class
- The **diamond problem** in other languages with multiple inheritance

  —— **Java not allow it**
- **Hierarchy depth matters: four or five levels** is almost always a design smell.

Pitfall Two: Composition Over Inheritance

- **Inheritance** (**is-a**): *Circle* is a *Shape*. Tight coupling.

  — child depends on the parent's internal structure; parent changes ripple through every subclass

- **Composition** (**has-a**): *PolyLine* has an *ArrayList*. Loose coupling.

  — your class only uses another class's public methods; internal changes in the helper class don't affect you as long as contracts are kept

---

When unsure, start with **composition**. It's safer.

```
// Inheritance (is-a)
class Circle extends Shape { ... }

// Composition (has-a)
class PolyLine {
    private ArrayList<Point> points;   // delegates to ArrayList
}
```

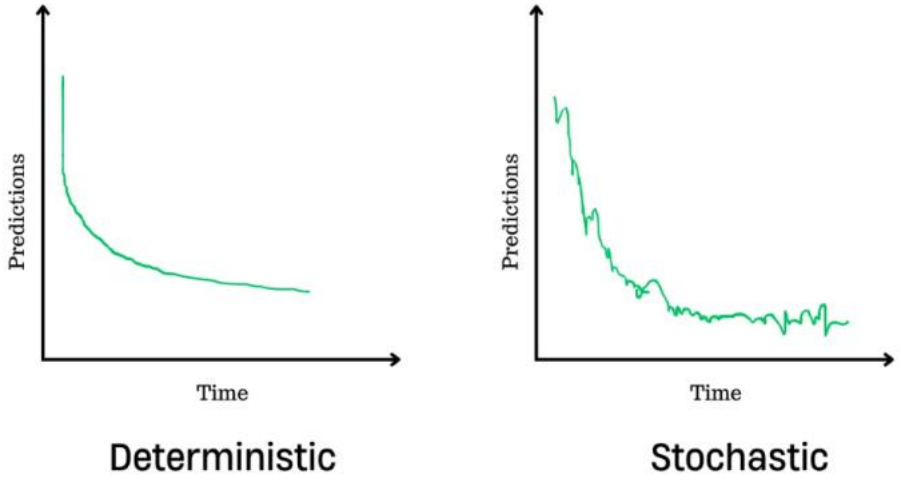
Why a Simulation Needs Randomness

Deterministic: same outcome every run

Stochastic: realistic variability

SIR needs random:

- Initial positions
- Infection rolls
- Movement direction



Asking the Random Class for Numbers

Math.random() — the one-liner

```
double r = Math.random();           // [0.0, 1.0)
int dice = (int)(Math.random() * 6) + 1;
```

- No object, no import
- Good for throwaway rolls, quick demos, single-shot scripts

---

new Random() — the reusable object

```
Random rng = new Random();

double x = rng.nextDouble() * width;        // ① position in area
boolean infect = rng.nextDouble() < p;       // ② probability roll
int idx = rng.nextInt(neighbors.size());     // ③ pick a neighbor
boolean coin = rng.nextBoolean();            //   fair true/false
double noise = rng.nextGaussian();           //   normal sample
```

Math.random() is essentially *new Random().nextDouble()* under the hood
— same source, different packaging

Reach for *Random* when you need a seed, integers, booleans, or Gaussians


```
Random rng1 = new Random();          // time-based seed → different every run
Random rng2 = new Random(42);       // fixed seed     → identical every run
```

- **No argument**

  → Java uses the current time as the seed

  → fresh sequence every run. Fine for a live demo, terrible for debugging

- **Fixed integer**

  → deterministic seed

  → exactly the same sequence every run. Identical input gives identical output, so you can rerun and reproduce a bug

Seeding Random for Reproducible Runs

WHAT YOU HAVE ON THE BENCH

`composition`

`inheritance + super(...)`

`overriding`

`instanceof`

`abstract class`

`Random + seed`

and from previous weeks

`enum` `static final` `ArrayList<T>`

Three classes are waiting:

Person · *Disease* · *Simulation*

Before you write a line, answer:

- Which class owns the randomness?
- What earns a *public static final*, what earns a subclass, what earns an *abstract* method?

Reading: Read the book chapters on inheritance and abstract classes

Coding

- **Reproducible random placement**. Every Person ends up at a random location inside the rectangular study area. Two back-to-back runs of your simulation should produce the same starting layout
- **Stochastic infection**. When an infectious person is within the infection radius of a susceptible person, the susceptible becomes infectious with the given probability
- **One place for the disease parameters.** The infection probability and the infection radius should live in one designated location (a *Disease* class)
- *More advanced*: support *multiple kinds of disease* (e.g. an airborne one and a contact-based one) whose transmission probabilities differ. And the generic *Disease* itself should not be something you can instantiate directly — there's no such thing as a "generic disease," only specific ones.

Submission: FirstNameLastNameW4.zip (*.java files only)

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

Exam Preview, Part A: Multiple Choice

Part A — Multiple Choice

Sample question — to give you a feel for the format.

---

1) Inside a child constructor, what does the call `super(args)` do?

a) Calls another constructor in the same class
b) Invokes the parent class constructor with the given arguments
c) Creates a brand new instance of the parent class
d) Is forbidden inside a constructor

$\approx$ *25 questions of this type · 2 pts each*

Exam Preview, Part B: Read Code, Predict Output

Part B — Read Code and Predict Output
Sample question.

1) What does the following program print?

```
class Watercourse {
    public String describe() { return "Watercourse"; }
}
class River extends Watercourse {
    @Override
    public String describe() {
        return super.describe() + " (river)";
    }
}

public class Demo {
    public static void main(String[] args) {
        Watercourse w = new River();
        System.out.println(w.describe());
    }
}
```

Output: _____

≈ *2 questions of this type · 10 pts each*

Part C — Write Code
Sample question.

1) A small environmental-monitoring system already has the following abstract class:

```
public abstract class Sensor {
    private String name;
    private Point  location;        // Point has getX(), getY()

    public Sensor(String name, Point location) {
        this.name = name;
        this.location = location;
    }

    public String getName()       { return name; }
    public Point  getLocation()   { return location; }

    public abstract String report();   // e.g. "Vienna-1: 21.3"
}
```

A **weather station** is a sensor that also records the current `temperature` (a double, in °C). Its `report()` should return the station's name, a colon and a space, and the current temperature — for example `"Vienna-1:  21.3"`.

Write the complete `WeatherStation` class.

≈ *2 questions of this type · 15 pts each*

The exam duration will be extended to 2 hours.

Code Example

1 RandomLocations

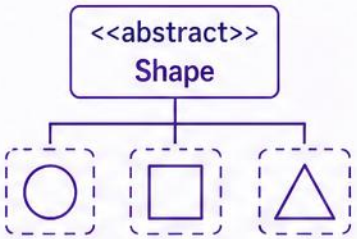
seeding, nextInt, nextDouble
in one place

2 Watercourse–River–Spring


```
super()
overriding()
instanceof
downcasting
```

super, overriding, instanceof,
downcasting end to end

3 Abstract Geometry hierarchy



abstract from declaration
to instantiation

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