Inheritance and Polymorphism in Java

Fundamental Object-Oriented Programming Mechanisms

Zbigniew Dendzik

2026

Lecture Plan

- Introduction to inheritance
- Inheriting from library classes
- Constructors in inheritance
- Polymorphism – concept and application
- Abstract classes
- Interfaces in Java
- Object serialization

Inheritance – Introduction

Inheritance – fundamental object-oriented programming mechanism. **Key concepts:**

- **Superclass** – base class
- **Subclass** – derived class
- **extends** – keyword defining inheritance

Benefits:

- Code reuse
- Class hierarchy organization
- Extending functionality of existing classes
- Enabling polymorphism

Inheritance Syntax

```
class SuperClass
{
    // fields and methods of superclass
}

class SubClass extends SuperClass
{
    // fields and methods of subclass
    // + everything inherited
}
```

Subclass inherits:

- All fields of superclass
- All methods of superclass
- Can add new fields and methods
- Can override superclass methods

Example: Inheriting from Rectangle

```
import java.awt.Rectangle;

class Prostokat extends Rectangle
{
    Prostokat(int a, int b)
    {
        super(a, b);
    }

    void info()
    {
        System.out.println(this);
    }
}
```

Prostokat class inherits from java.awt.Rectangle:

- Fields: x, y, width, height

Using the Inherited Class

```
public class Program
{
    public static void main(String[] args)
    {
        Prostokat a = new Prostokat(3, 4);
        a.info();

        Prostokat b = new Prostokat(2, 2);
        b.info();

        if(a.intersects(b))
        {
            System.out.println("-- they intersect --");
        }

        a.translate(5, 3);
        a.info();
    }
```

Which Methods Are Inherited?

In Prostokat class:

- `info()` – implemented locally

Inherited from java.awt.Rectangle:

- `intersects(Rectangle r)` – do rectangles intersect?
- `translate(int dx, int dy)` – shift by vector
- `contains(int x, int y)` – is point inside?
- `setLocation(int x, int y)` – set position
- `toString()` – text representation

Constructors and super()

Important rules:

- Constructors are **NOT** inherited
- Subclass must call superclass constructor
- `super()` – superclass constructor invocation
- Call to `super()` must be first statement

Example:

- `super(a, b);` – calls Rectangle(int w, int h) constructor
- If we don't call super() explicitly, compiler adds no-arg super()
- If superclass has no no-arg constructor $\rightarrow$ compilation error

Constructor with Point

```
import java.awt.Rectangle;
import java.awt.Point;

class Prostokat extends Rectangle
{
    Prostokat(int a, int b)
    {
        super(a, b);
    }

    Prostokat(Point vertex, int length, int width)
    {
        super(vertex.x, vertex.y,
              length, width);
    }

    void info()
    {
```

Adding Custom Methods

```
class Prostokat extends Rectangle
{
    Prostokat(int a, int b)
    {
        super(a, b);
    }

    boolean adjacent(Prostokat other)
    {
        // Check if rectangles are adjacent
        // (intersect or share an edge)

        Rectangle expanded = new Rectangle(this);
        expanded.grow(1, 1);

        return expanded.intersects(other) &&
               !this.intersects(other);
    }
```

Polymorphism – Introduction

Polymorphism – ability to treat objects of different classes uniformly. **Key features:**

- Superclass reference can point to subclass object
- Method invocation depends on actual object type (not reference type)
- Enables writing more general code
- Foundation of flexible object-oriented systems

Applications:

- Arrays of objects of different types
- Method parameters
- Heterogeneous collections

Abstract Classes

```
abstract class Shape
{
    abstract double area();
    abstract double perimeter();

    void info()
    {
        System.out.println(this);
    }
}
```

Abstract class:

- Marked with `abstract` keyword
- **Cannot** instantiate it
- Can contain abstract methods (without implementation)
- Can contain concrete methods (with implementation)
- Serves as template for subclasses

Circle Class Implementation

```
class Circle extends Shape
{
    double radius;

    Circle(double radius)
    {
        this.radius = radius;
    }

    double area()
    {
        return 3.14 * radius * radius;
    }

    double perimeter()
    {
        return 2 * 3.14 * radius;
    }
```

Rectangle Class Implementation

```
class Rectangle extends Shape
{
    double length;
    double width;

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

    double area()
    {
        return length * width;
    }

    double perimeter()
    {
```

Polymorphism in Action

```
public class Program
{
    public static void main(String[] args)
    {
        Shape z = new Circle(2);
        z.info();

        Shape[] a = {new Rectangle(3,5), new Circle(8), new
            Circle(3)};

        Shape x;
        double sum = 0;

        for(int i = 0; i < a.length; i++)
        {
            x = a[i];
            x.info();
            sum = sum + x.area();
```

Polymorphism – Mechanism

What happens in the example:

- `Shape[] a` – array of references to Shape
- Elements are Rectangle and Circle objects
- `x.info()` – method called from Shape class
- `x.area()` – method called from Rectangle or Circle
- Decision about method at runtime (late binding)

Benefits:

- Single code handles different types
- Easy to add new shapes
- Flexibility and extensibility

Interfaces in Java

Interface – set of method declarations without implementation. **Interface**

characteristics:

- Defined with `interface` keyword
- Contain only method signatures (no bodies)
- Classes `implement` interfaces (`implements` keyword)
- Class can implement multiple interfaces
- Interface defines a "contract" – set of methods to implement

Differences from inheritance:

- Inheritance: one superclass
- Interfaces: multiple interfaces

Interface Definition

```
interface Searchable
{
    boolean matches(String pattern);
}

abstract class Document implements Searchable
{
    // Abstract class implementing interface
    // Subclasses must implement matches()
}
```

Notes:

- Abstract class can implement interface without implementing methods
- Concrete subclasses must provide implementations
- Interface defines common behavior

Document Classes Implementation

```
class Passport extends Document
{
    String number;
    String firstName;
    String lastName;

    public boolean matches(String pattern)
    {
        return firstName.equalsIgnoreCase(pattern) ||
               lastName.equalsIgnoreCase(pattern);
    }

    public String toString()
    {
        return "Passport: " + firstName + " " + lastName +
               ", no: " + number;
    }
}
```

ID Card Implementation

```
class IDCard extends Document
{
    String number;
    String firstName;
    String lastName;

    public boolean matches(String pattern)
    {
        return firstName.equalsIgnoreCase(pattern) ||
               lastName.equalsIgnoreCase(pattern) ||
               number.equalsIgnoreCase(pattern);
    }

    public String toString()
    {
        return "ID Card: " + firstName + " " + lastName +
               ", no: " + number;
    }
```

Database Searching

```
public class Program
{
    public static void main(String[] args)
    {
        Document[] database = {
            new Passport(),
            new IDCard(),
            new Passport()
        };

        Document z;
        String pattern = "Smith";

        for(int i = 0; i < database.length; i++)
        {
            z = database[i];
            if(z.matches(pattern))
                System.out.println("found: " + z);
```

Object Serialization

Serialization – converting object to byte stream. **Applications:**

- Saving objects to file
- Sending objects over network
- Persistent storage of application state

Serializable Interface:

- `java.io.Serializable`
- Marker interface – no methods
- Class must implement to enable serialization
- All fields must be serializable

Serializable Classes

```
import java.io.*;

class Person implements Serializable
{
    String firstName;
    String lastName;
    int birthYear;

    Person(String firstName, String lastName, int birthYear)
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.birthYear = birthYear;
    }

    public String toString()
    {
        return this.firstName + " " + this.lastName +
```

Serializable IDCard Class

```
class IDCard implements Serializable
{
    Person owner;
    String number;

    IDCard(Person owner, String number)
    {
        this.owner = owner;
        this.number = number;
    }

    public String toString()
    {
        return "<id:> " + owner.toString() +
               " " + this.number;
    }

    void info()
```

Writing Object to File

```
IDCard z = new IDCard(
    new Person("John", "Smith", 1980),
    "ABC123456"
);

z.info();

try
{
    ObjectOutputStream outp = new ObjectOutputStream(
        new FileOutputStream("file.dat"));

    outp.writeObject(z);
    outp.close();

    System.out.println("Object saved!");
}
catch(Exception e)
```

Reading Object from File

```
try
{
    ObjectInputStream inp = new ObjectInputStream(
        new FileInputStream("file.dat"));

    Object o = inp.readObject();
    IDCard x = (IDCard)o;

    inp.close();

    x.info();
}
catch(FileNotFoundException e)
{
    System.out.println("File does not exist!");
}
catch(Exception e)
{
```

Exception Handling in Serialization

Possible exceptions:

- `FileNotFoundException` – file does not exist
- `IOException` – read/write error
- `ClassNotFoundException` – class definition missing
- `NotSerializableException` – class not serializable

Best practices:

- Catch specific exceptions
- Provide user messages
- Check file existence before reading
- Always close streams (close())

Advanced Applications

Example projects:

1. **Banking account system**
   - Abstract Account class
   - Interfaces: Interest, Tax
   - Concrete classes: SavingsAccount, Deposit
   - Simulation of capitalization and fees
2. **Geometric shapes hierarchy**
   - Adding: Triangle, Trapezoid, Rhombus
   - Calculating areas and perimeters
   - Summing shape parameters
3. **Document database**
   - Extension with new document types
   - Multi-criteria search
   - Database serialization to file

Summary

- **Inheritance** – extends, super(), method overriding
- **Constructors** – not inherited, super() invocation
- **Polymorphism** – uniform treatment of different types
- **Abstract classes** – templates with abstract methods
- **Interfaces** – contracts defining behaviors
- **Serialization** – saving objects to file
- **OOP** – code reuse, flexibility