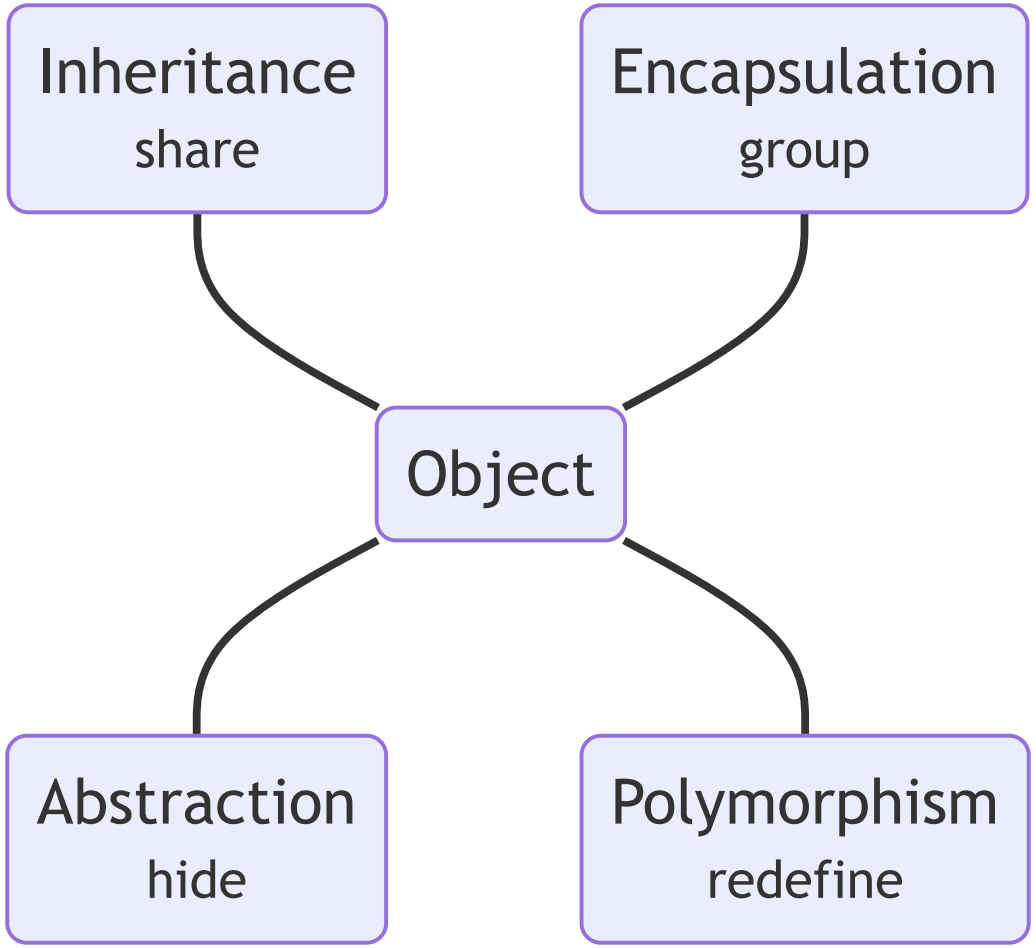
SE 350 - Object-Oriented Software Development

Object-Oriented Programming Principles: Inheritance

Instructor: Stefan Mitsch

Principles of Object-Oriented Programming



Learning Objectives

- Understand what gets inherited and what can be changed
- Understand the difference between overloading and overriding
- Understand why inheriting from multiple base classes is problematic
- Build a first intuition about UML class diagrams

Inheritance

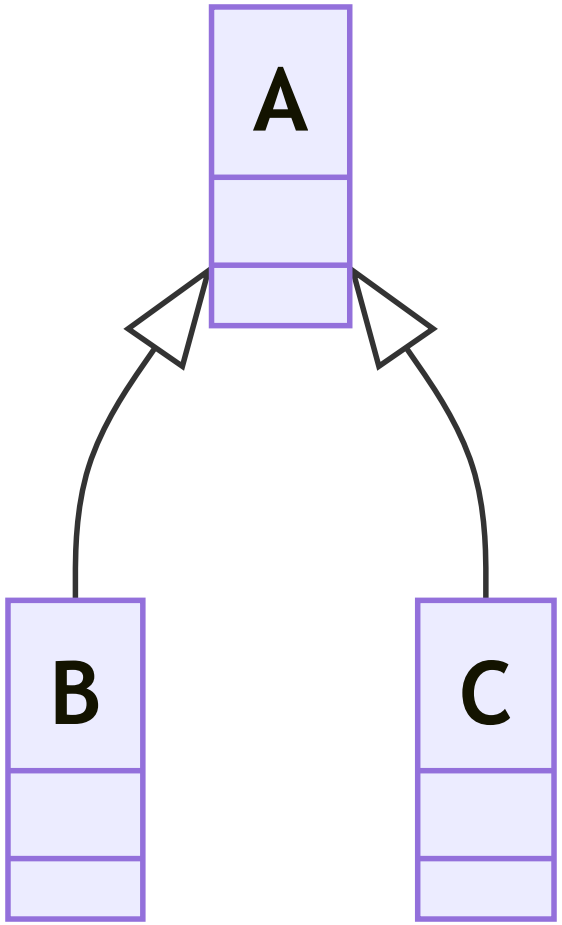
- Inherit state and behavior
- Define commonalities in super-class, specialize in sub-class
- Keyword `extends`
- Sub-class inherits all members, but has access only to `protected` and `public`
- Decide inheritance based on **is-a** relationship

```java
public class Employee {
  private int id;
  public void print() { /* ... */ }
}

public class Manager extends Employee {
  private List<Employee> reportees;
}
```

Class Hierarchies

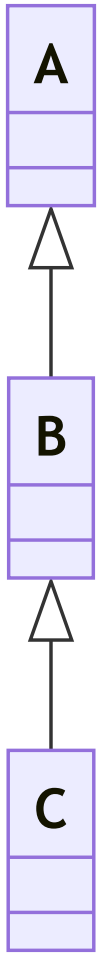
UML Class Diagram



Corresponding Java Code

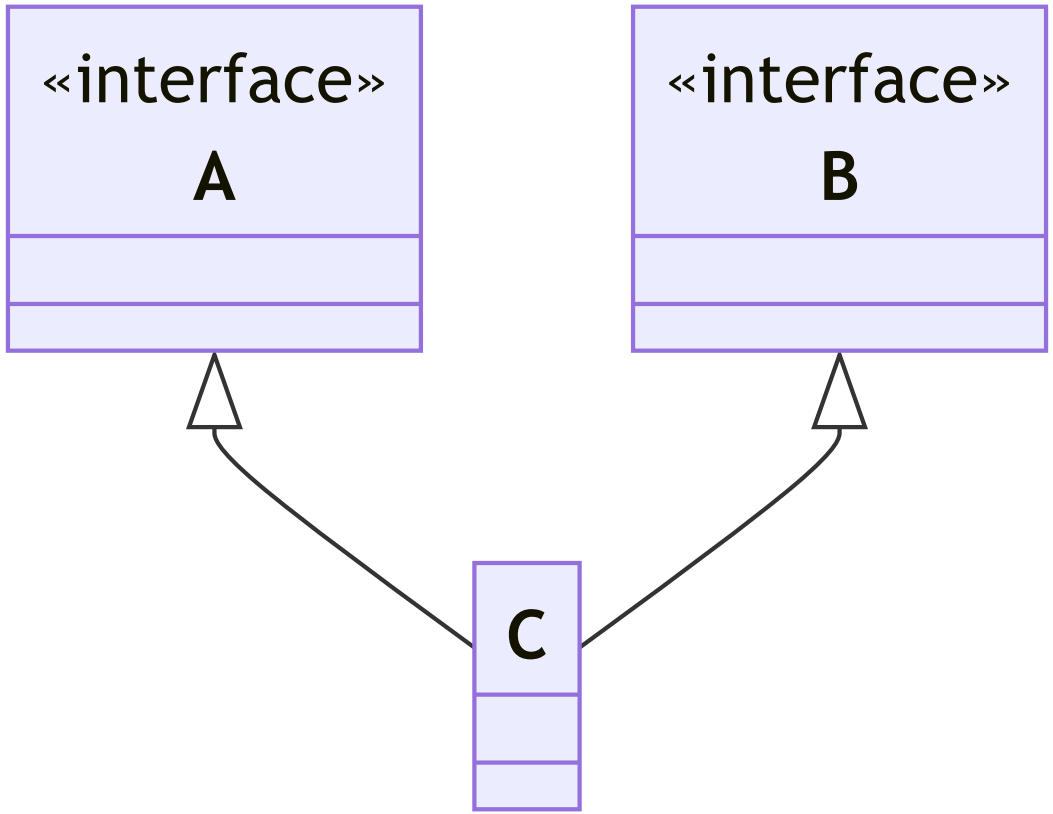
```
public class A {}
public class B extends A {}
public class C extends A {}
```

Class Hierarchies




```
public class A {}
public class B extends A {}
public class C extends B {}
```

Class Hierarchies




```
public interface A {}
public interface B {}
public class C implements A, B {}
```

Java Class Hierarchy

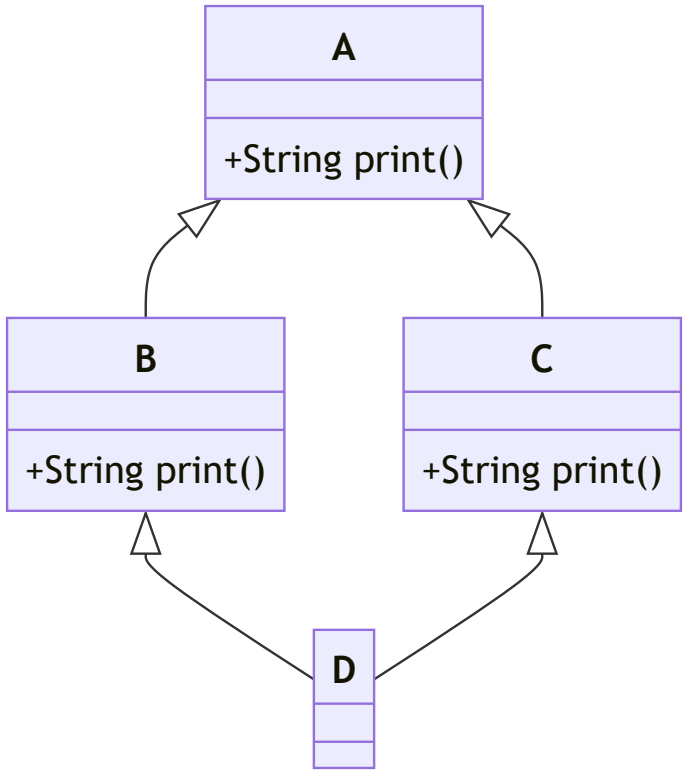
- `java.lang.Object` is at the root of the class hierarchy
- Primitive and complex datatypes are not in a uniform class hierarchy (boxing and unboxing, e.g., `int` and `Integer`)
- Inheritance is transitive
- Sub-class can add but not remove members
- Private members are inherited but inaccessible

```
public class A {
  private int a;
  public int getA() { return a; }
  public int setA(int a) { this.a = a; }
}
public class B extends A {}

public class Demo {
  public static void main(String[] args) {
    B b = new B();
    System.out.println("b.a = " + b.a); // compile error
    System.out.println("a.getA() = " + b.getA());
    b.setA(5);
    System.out.println("b.getA() = " + b.getA());
  }
}
```

Multiple Inheritance of Classes

- Prohibited in Java: diamond problem
- Allowed in other languages (e.g., C++ does not distinguish between classes and interfaces; requires programmers to use appropriately)




```
public class A {
  public String print() { return "A"; }
}
public class B extends A {
  public String print() { return "B"; }
}
public class C extends A {
  public String print() { return "C"; }
}
public class D extends B, C {}
```

- What should be the behavior of `print()`?

Constructor Sequence

- Constructors are executed along inheritance chain (first super-class constructor, then sub-class constructor)
- Default-constructor if not specified explicitly
- Explicit call must be first in constructor

```
public class A {
  public A() { System.out.println("A"); }
}
public class B extends A {
  public B(int i) { System.out.println("B-" + i); }
}
public class C extends B {
  public C(int i) {
    super(i);
    System.out.println("C");
  }
}
public class Demo {
  public static void main(String[] args) {
    C c = new C(3);
  }
}
```

Accessing Inherited Members

- Call a super-class constructor explicitly: `super(...)` must be first statement in sub-class constructor (implicit call of default-constructor if omitted)
- Access non-private fields by name
- Static type disambiguates shadowed fields
  - `this` : static type of current class
  - `super` : static type of super-class
- Dynamic type determines method

```
public class Employee {
  public long id;
  public void print() { System.out.println("E-" + this.id); }
}
public class Manager extends Employee {
  public long id; // shadows Employee.id
  public Manager() {
    super(); // must be first statement
    // other initialization
    this.id = 20L;  // static type Manager
    super.id = 10L; // static type Employee
  }
  @override
  public void print() { System.out.println("M-" + this.id); }
}

public class Demo {
  Manager m = new Manager();
  // m has static type Manager -> access id in Manager
  System.out.println(m.id);
  Employee e = m;
  // e has static type Employee -> access id in Employee
  System.out.println(e.id);
  // both e and m have dynamic type Manager
  // print of Manager is executed both times
  m.print();
  e.print();
}
```

Constructor Chaining

- If used, `super` or `this` must be first statement
- Cannot mix `this` and `super` in a single constructor (would result in duplicate calls to potentially different super-class constructors)
- Properly chain constructors to minimize code duplication

```
public class Employee {
  private int id;
  public Employee() { this(5); }
  public Employee(int id) { this.id = id; }
}
public class Manager extends Employee {
  private List<Employee> reportees;
  public Manager() { this(2); }
  public Manager(int id) { this(id, new List()); }
  public Manager(int id, List<Employee> reportees) {
    super(id);
    this.reportees = reportees;
  }
}
```

Summary

- Inheritance: inherit all members of a super-class, but only non-private fields are accessible
- `super` and `this` have different static type to disambiguate shadowed names
- Methods are selected based on the dynamic type of an object
- Avoid initialization duplication with proper constructor chaining