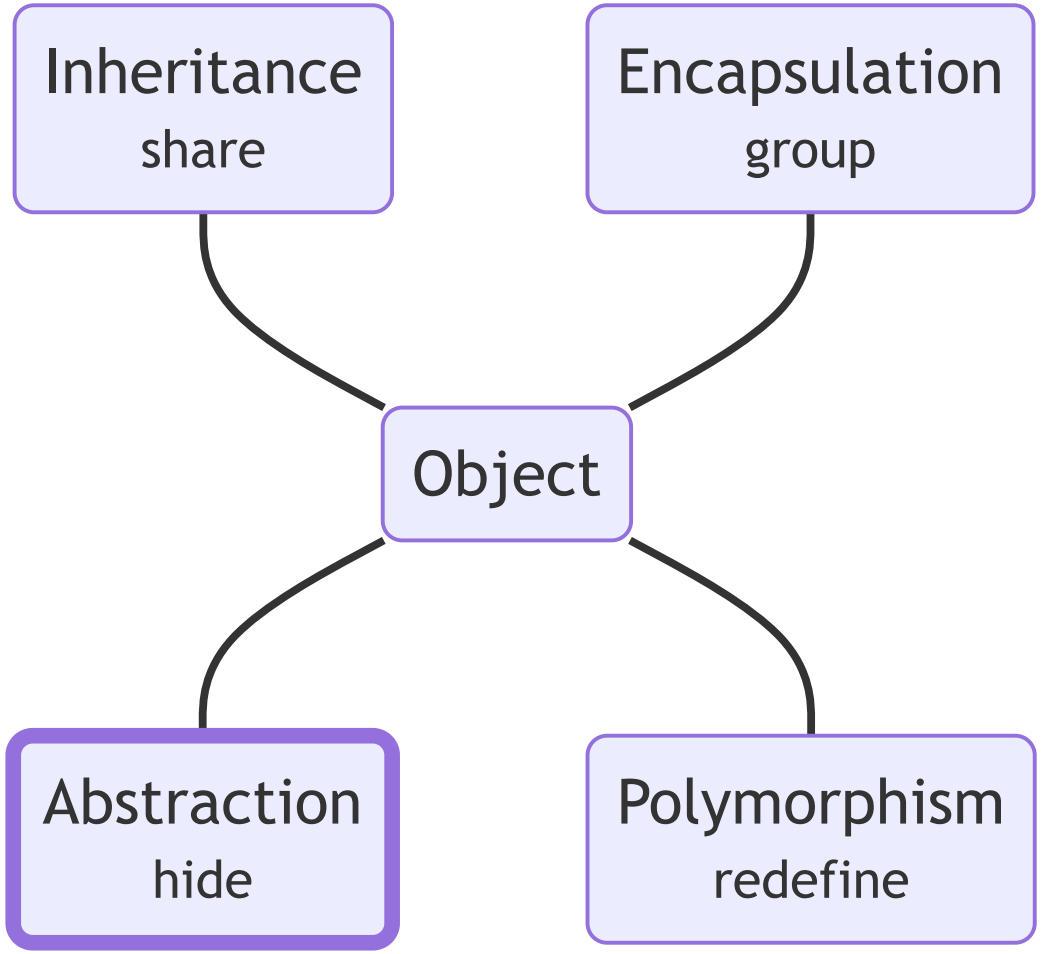
SE 350 - Object-Oriented Software Development

Object-Oriented Programming Principles: Abstraction

Instructor: Stefan Mitsch


Inheritance
share
Encapsulation
group
Object
Abstraction
hide
Polymorphism
redefine

Learning Objectives

- Understand abstract classes
- Understand interfaces
- Understand design guidelines interface vs. abstract class

Abstraction

- Provide an incomplete blueprint
- Abstract class: logical concept to categorize and provide common (incomplete) functionality, cannot be instantiated (e.g., `Animal`)
- Abstract method: declaration without implementation
- Concrete sub-classes provide missing implementations
  - Must implement `abstract` methods
  - May override non-final methods
  - Cannot override final methods

```
// abstract class cannot be instantiated
public abstract class Animal {
  // specify interface but not implementation
  public abstract String sound();
}

// can create a hierarchy of abstract classes
public abstract class Pet extends Animal {
  private String name;
  public Pet(String name) { this.name = name; }
  public final String getName() { return name; }
  public final void setName(String name) { this.name = name; }

  public String getTag() {
    return doGetTag() + ": " + getName();
  }
  protected abstract String doGetTag();
}

// concrete class provides implementations
public final class Dog extends Pet {
  public Dog(String name) { super(name); }

  @Override
  public String sound() { return "bark"; }

  @Override
  protected final String doGetTag() { return "Collar Tag"; }
}
```

Abstract Classes

- Can contain fields, abstract methods, and concrete methods
- Can use any access modifiers
  (but cannot declare `private abstract` methods; why?)
- Any class with one or more `abstract` methods must be declared `abstract`
- Any concrete class must implement all `abstract` methods
- Cannot create `abstract final` classes; why?
- Constructors cannot be `final` or `abstract` or `static`; why?

Modifier Compatibility

- Modifiers in sub-classes must be compatible with modifiers in super-classes; why? what is compatible?

```
public abstract class Animal {
  protected abstract String sound();
}

public class Dog extends Animal {
  // which modifiers are compatible?
  private
  protected
  public    String sound() { return "bark"; }
}
```

Interfaces

- Logical concept to categorize the signatures of common functionality
- Contains method signatures, but no implementations
- All methods are implicitly `public`
- All fields are implicitly `public static final`
- Declares **what** to implement, not how to implement
- Classes can inherit only from at most one super-class, but can implement multiple interfaces

```java
public interface Walkable {
  void walk(int distance);
}

public interface Swimmable {
  void swim(int distance);
}

public abstract class Animal {
  // common animal functionality
}

public class Dog
  extends Animal
  implements Walkable, Swimmable {
  public void walk(int distance) { /* implementation */ }
  public void swim(int distance) { /* implementation */ }
}
```

Abstract Classes vs. Interfaces

- Use abstract classes to provide implementation stubs
- Use interfaces to declare common signatures
- Extend and implement
  - Abstract class can extend another class and implement interfaces (but bad design to have abstract class extend a concrete class)
  - Interface can extend other interfaces
- Why does the diamond problem not occur with interfaces?

Default Methods in Interfaces

- Common idiom in (old) Java: interface + abstract base class with default implementation of interface
- Since Java 8: `default` methods in interface
- Can override default methods in concrete classes
- Avoid diamond problem: must override if multiple default methods

```java
public interface Walkable {
  void walk(int distance);
  default void stopped() { walk(0); }
}
public interface Swimmable {
  void swim(int distance);
  default void stopped() { swim(0); }
}
public class Dog implements Walkable, Swimmable {
  @Override
  public void stopped() {
    walk(0); // or: Walkable.super.stopped()
    swim(0); // or: Swimmable.super.stopped()
  }
}
```

Summary

- Abstract classes provide common data and common (partial) implementation
  - Can extend another class
  - Can implement many interfaces
- Interfaces specify common functionality signatures
  - Can define constants (`public static final` by default)
  - Can define default implementations (`default`)
  - Can extend other interfaces