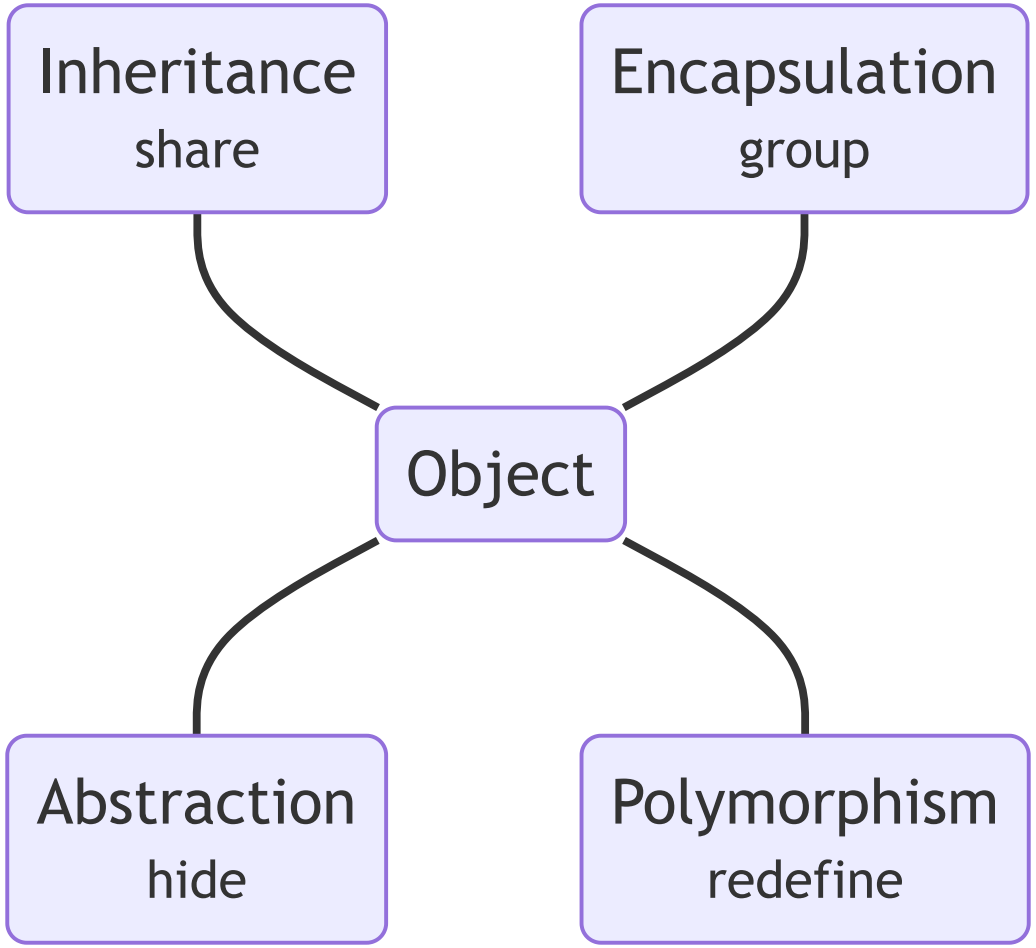
SE 350 - Object-Oriented Software Development

Object-Oriented Programming Principles: Polymorphism

Instructor: Stefan Mitsch

Principles of Object-Oriented Programming



Learning Objectives

- Understand method overloading
- Understand method overriding
- Understand polymorphism
- Understand the difference between static binding/static dispatch and dynamic binding/dynamic dispatch

Method Overloading

- Methods with same name are disambiguated by their arguments (type, number)
- Static binding: method is selected at compile-time
- Methods only differing in return-type are disallowed
- Constructors can be overloaded the same way as methods

```
public class Adder {
  public int sum(int x, int y) {
    return x+y;
  }
  public int sum(int x, int y, int z) {
    return x+y+z;
  }
  public double sum(double x, double y) {
    return x+y;
  }
  public String sum(String x, String y) {
    return x.concat(y);
  }

  public static void main(String[] args) {
    Adder oa = new Adder();
    oa.sum(1, 2);          // calls sum(int,int)
    oa.sum("De", "Paul")  // calls sum(String,String)
  }
}
```

Polymorphism

- One name with many forms
- **Static binding**: static dispatch at compile-time, compiler selects method based on the static type of a reference (in Java for method overloading and `static` methods!)
- **Dynamic binding**: dynamic dispatch at runtime, dynamic type of a reference determines which method to execute

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

public class Dog extends Animal {
  public String sound() { return "bark"; }
}

public class Cat extends Animal {
  public String sound() { return "meow"; }
}

public class Demo {
  public static void main(String[] args) {
    // a has static type Animal, dynamic type Dog
    Animal a = new Dog();
    System.out.println(a.sound());
    // a has static type Animal, dynamic type Cat
    a = new Cat();
    System.out.println(a.sound());
  }
}
```

Static Dispatch for `static` Methods

- `static` methods are always statically dispatched based on the static type of a reference/class
- Cleaner way of resolving `static` methods is directly through the class name: prefer `Parent.clazzStatic();` over `Parent p; p.clazzStatic();`

```
public class Parent {
  public static String clazzStatic() { return "Parent"; }
  public String clazzDynamic() { return "Parent"; }
}

public class Child extends Parent {
  public static String clazzStatic() { return "Child"; }
  @override
  public String clazzDynamic() { return "Child"; }
}

public class Demo {
  Parent a = new Child();
  System.out.println(a.clazzStatic());
  System.out.println(a.clazzDynamic());
  Child b = new Child();
  System.out.println(b.clazzStatic());
  System.out.println(b.clazzDynamic());
}
```

Method Overriding

- Change behavior of super-class
- Define extension points in super-class, implement in sub-class
- Arguments, return types, and modifiers must be compatible

```
public class Parent {
  public void print() { System.out.println(doPrint()); }
  public String doPrint() { return "Parent"; }
}

public class Child extends Parent {
  public String doPrint() { return "Child"; }
}

public class Demo {
  public static void main(String[] args) {
    Child c = new Child();
    c.print();

    Parent p = c;
    p.print();
  }
}
```

Upcasting and Downcasting

- Upcasting: refer to an object through a reference statically typed using a super-type
- Downcasting: refer to an object through a reference statically typed using a sub-type
- Upcasting is always safe (all dogs are animals)
- Downcasting may fail (only some animals are dogs)
- Always do an **is-a** test

```
public class Parent {
  public static String clazzStatic() { return "Parent"; }
  public String clazzDynamic() { return "Parent"; }
}

public class Child extends Parent {
  public static String clazzStatic() { return "Child"; }
  @override
  public String clazzDynamic() { return "Child"; }
}

Child a = new Child();
System.out.println(a.clazzStatic());
System.out.println(a.clazzDynamic());

Parent b = a; // Upcast is always safe
System.out.println(b.clazzStatic());
System.out.println(b.clazzDynamic());

// downcast requires explicit cast
// is only safe here because dynamic type of b is Child
Child c = (Child)b;
System.out.println(c.clazzStatic());
System.out.println(c.clazzDynamic());
```

Prevent Overriding

- Some behavior may be designed unchangeable
- Preserve consistent state of an object
- `final` prevents overriding
  - `final` class: prevent inheritance (alternative: private constructors)
  - `final` method: prevent overriding
  - `final` variable: prevent changing
  - `final` constructor: not allowed (constructors are never inherited)

```
// final class cannot be extended
public final class Parent { }
// compile error
public class Child extends Parent { }
```

```
public class Parent {
  public final String print() { return "Parent"; }
}
public class Child extends Parent {
  // compile error
  public String print() { return "Child"; }
}
```

```
public class Consts {
  // value of final variable cannot be changed
  public static final int ID = 5;
  public static void main(String[] args) {
    // compile error
    Consts.ID = 2;
  }
}
```

No Overriding of Private Methods

- Private methods cannot be overridden (are implicitly final because inaccessible from sub-class)

```
public class Parent {
  private String print() { return "Parent"; }
}
public class Child extends Parent {
  // adds a separate private print method
  private String print() { return "Child"; }
}
```

Subtype Polymorphism and Containers

Linked list for each content type

```
public class Node {
  private int data;
  private Node next;
  // ...
}
```

- Type safety
- ⚡ Code duplication

Generic linked list

```
public class Node {
  private Object data;
  private Node next;
  // ...
}
```

- No code duplication
- ⚡ No static protection against casts from `Object`

Parametric Polymorphism: Generics

```
public class Node<X> {
  private X data;
  private Node<X> next;
  // ...
}
```

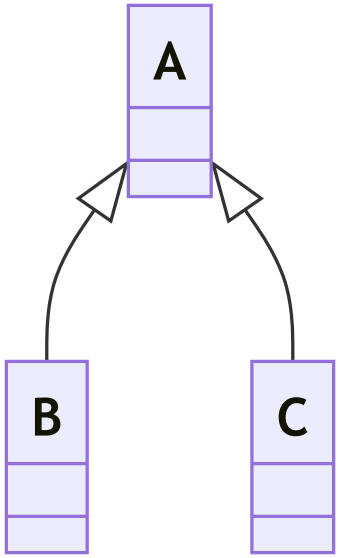
- Type safety
- No code duplication

Challenges with Java Generics

- Unclean Java language design (backwards compatibility)
- Type parameters not stored at runtime: cannot instantiate `X` or `X[]`
- Have to use careful unchecked casts

```
public class ArrayList<X> {
  private X[] items;
  // public ArrayList(int n) { items = new X[n]; } // compile error
  @SuppressWarnings("unchecked")
  public ArrayList(int n) { items = (X[]) new Object[n]; }
  public void put(int i, X item) { items[i] = item; }
  public X    get(int i)         { return items[i]; }
}
```

Is-A Test for Parametric Polymorphism



- `B extends A`: a `B` is an `A`
- `C extends A`: a `C` is an `A`
- A `B` is not a `C` (sets are disjoint)
- An `A` is not necessarily a `B` (some `A`s are `C`s)

❓ From the subtype relationship on the left, does any of the subtype relationship between containers below follow?

- A `List<B>` is a `List<A>`?
- A `List<A>` is a `List<B>`?

Subtyping

```
public class A {}
public class B extends A {}
A a = new A();
A b = new B(); // ok, B is a subtype of A
```

- `B` is a subtype of `A`: set of all `B` instances is a subset of set of all `A` instances
- Subtype relationship `B <: A` has property `x:B` and `B <: A` then `x:A`
- Java has a top-type: `Object`

Subtyping and Parametric Polymorphism

- `B <: A` then `List<B> <: List<A>`?

```
class A {}
class B extends A {}

List<B> bs = new ArrayList<>();
List<A> as = bs; // should this be allowed?
```

- ⚡ `List` is writeable, can violate type promise of `bs`!

```
as.add(new A()); // as and bs refer to the same list
B b1 = bs.get(0);
B b2 = bs.get(1);
```

Subtyping and Parametric Polymorphism

- Wildcards to define upper type bounds

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

List<B> bs = new ArrayList<>();
List<? extends A> as = bs;

as.add(new A()); // allowed?
as.add(new B()); // allowed?
as.add(null);    // allowed?

List<C> cs = new ArrayList<>();
as = cs;
```

- `List<? extends A> xs`: a read-only list that guarantees to provide `A` instances

Subtyping and Parametric Polymorphism

- Wildcard to define lower subtype bound

```
class A {}
class B extends A {}

List<B> bs = new ArrayList<>();
List<? super B> xs = bs;
xs.add(new A()); // allowed?
xs.add(new B()); // allowed?

List<A> as = new ArrayList<>();
xs = as;
xs.add(new A()); // allowed?
xs.add(new B()); // allowed?
```

- `List<? super B> xs` : a writeable list that guarantees to accept `B`

Summary

- Method overloading: compiler selects a method statically based on the method arguments
- Method overriding: change behavior of inherited methods, dynamic dispatch at runtime selects method based on the dynamic type of an object
- Parametric polymorphism: parameterize containers with element types

Exercises

- What is the output of the code below and why?
- Which methods are statically dispatched, which ones dynamically dispatched?

```
public abstract class Animal {

}

public class Dog extends Animal {

}

public class Cat extends Animal {

}
```

```
public class Printer {
  public void print(Animal a) {
    System.out.println("I am an animal");
  }
  public void print(Dog d) {
    System.out.println("I am a dog");
  }
  public void print(Cat c) {
    System.out.println("I am a cat");
  }
  public static void main(String[] args) {
    Printer p = new Printer();

    Cat c = new Cat();
    p.print(c);
    p.print((Animal)c);

    Dog d = new Dog();
    p.print(d);
    Animal a = d;
    p.print(a);
  }
}
```

Exercises

- What is the output of the code below and why?
- Which methods are statically dispatched, which ones dynamically dispatched?

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

public class Dog extends Animal {
  public String sound() { return "bark"; }
}

public class Cat extends Animal {
  public String sound() { return "meow"; }
}
```

```
public class Printer {
  public void print(Animal a) {
    System.out.println("I am an animal: " + a.sound());
  }
  public void print(Dog d) {
    System.out.println("I am a dog: " + d.sound());
  }
  public void print(Cat c) {
    System.out.println("I am a cat: " + c.sound());
  }
  public static void main(String[] args) {
    Printer p = new Printer();

    Cat c = new Cat();
    p.print(c);
    p.print((Animal)c);

    Dog d = new Dog();
    p.print(d);
    Animal a = d;
    p.print(a);
  }
}
```

Exercises

- Parametric polymorphism: answer for each line whether it is allowed in Java, and what can be done with such a list reference.

```
List<Number> xs = new ArrayList<Number>();

List<Number> xs = new ArrayList<Integer>();

List<? extends Number> xs = new ArrayList<Integer>();

List<? super Number> xs = new ArrayList<Integer>();

List<? super Integer> xs = new ArrayList<Number>();
```