Object-Oriented Programming

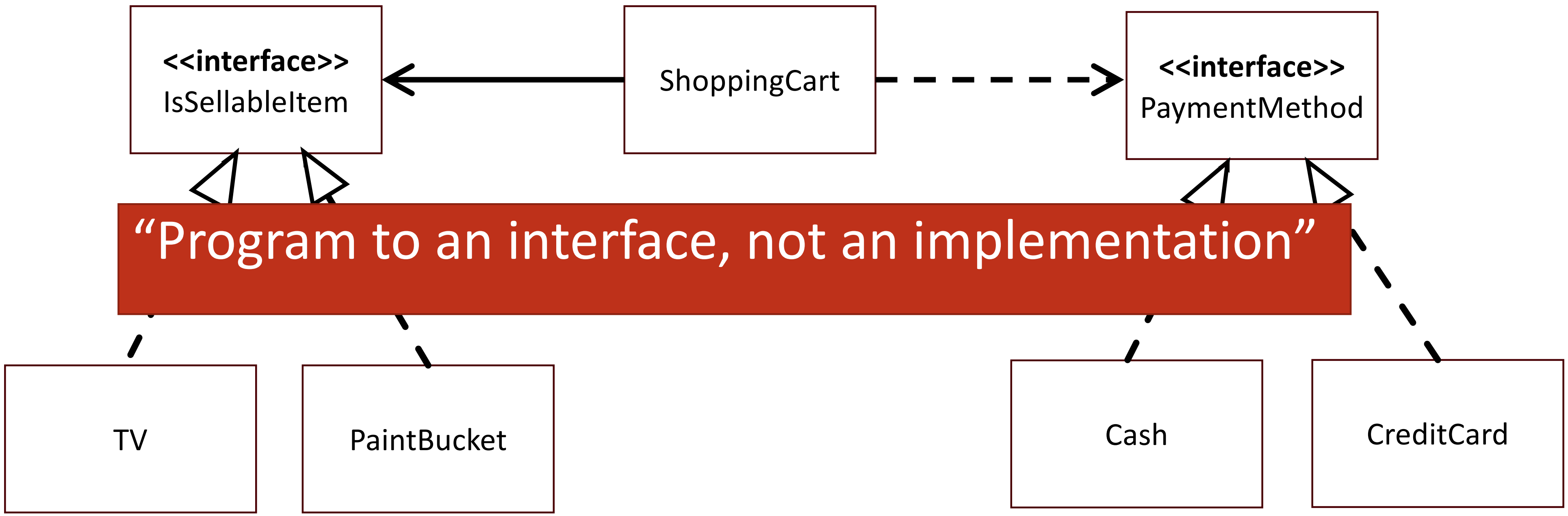
Lecture 3: Interfaces, part 2

Sebastian Junges

Interfaces

- Specify (visible, public) behavior
- Can be thought of contracts:
  A class implementing an interface must provide
  the behavior specified by the interface

Class Diagram



Goal Today

After today's lecture, you should be able to explain

- Default methods
- Polymorphism, early and late binding
- Composition
- Creating objects without explicit types / static factory methods
- Markers

Implementing multiple interfaces

```java
public interface EnergyConsumer {
    int getPower();
    int getRequiredVoltage();
}
```

Default methods

- Key idea: avoid code duplication!

Default Methods

In addition to public method declarations, interfaces can have

- default public method implementations
- static method implementations
- private method implementations

Old texts on Java do not discuss this. It was introduced in Java 8 (2014).

Default methods do not need to be overridden

```java
public interface EnergyConsumer {
    int getPower();
    default int getRequiredVoltage() { return 230; }
}

public class TV implements EnergyConsumer {
    @Override
    public int getPower() { return 300; }
}
```

Default methods can be overridden

```java
public interface EnergyConsumer {
    int getPower();
    default int getRequiredVoltage() { return 230; }
}

public class Heater implements EnergyConsumer {
    @Override
    public int getPower() { return 300; }

    @Override
    public int getRequiredVoltage() { return 380; }
}
```

Default methods

```
public static void main(String[] args) {
    Heater h1 = new Heater();
    System.out.println(h1.getRequiredVoltage());
    TV tv = new TV(32);
    System.out.println(tv.getRequiredVoltage());
    EnergyConsumer h2 = new Heater();
    System.out.println(h2.getRequiredVoltage());
}
```

```
Compile & Run Lecture3:

380
230
380

Success
```

Goal Today

After today's lecture, you should be able to explain

- Default methods
- Polymorphism, early and late binding
- Composition
- Creating objects without explicit types / static factory methods
- Markers

Recap: (Object) Polymorphism

- Poly – many
- Morph – form

Objects can take multiple forms or types.

- The TV is an item with a price
- Also an item with energy consumption

Recap: Static vs Dynamic Types

```java
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    IsSellableItem item;
    if (scanner.nextInt() > 10) {
        item = new TV(32);
    } else {
        item = new PaintBucket(25);
    }
    System.out.println(item);
    System.out.println(item.getPrice());
}
```

If we input a number > 10, then item holds a TV, otherwise item holds a PaintBucket

at runtime.

Item is an IsSellableItem. We cannot know whether it refers to a TV or a PaintBucket

at compile time.

Recap: Dynamic types

The static type is determined by the declaration of the variable / the field / the parameter. At compile time, we always know the static type.

The dynamic type is always a subtype of the static type. Only during executing, we know the type it holds.

Binding: Which methods to call

- Objects / variables have multiple types.
  Ambiguity which methods to call.

- Early binding: Use the static type, decide at compile time
- Late binding: Use the dynamic type, decide at runtime

Late binding of `this`

```
public static void main(String[] args) {
    Heater h1 = new Heater();
    System.out.println(h1.getRequiredVoltage());
    TV tv = new TV(32);
    System.out.println(tv.getRequiredVoltage());
    EnergyConsumer h2 = new Heater();
    System.out.println(h2.getRequiredVoltage());
}
```

```
Compile & Run Lecture3:

380
230
380

Success
```

Early binding of parameters (1)

```
public static void whatAmI(EnergyConsumer ec) {
    System.out.println("an Energyconsumer");
}

public static void whatAmI(TV tv) {
    System.out.println("a TV");
}

public static void whatAmI(Heater h) {
    System.out.println("a Heater");
}
```

```
public static void main(String[] a) {
    Heater h1 = new Heater();
    whatAmI(h1);
    TV tv = new TV(32);
    whatAmI(tv);
    EnergyConsumer h2 = new Heater();
    whatAmI(h2);

}
```

```
Compile & Run Lecture3:

a Heater
a TV
an EnergyConsumer

Success
```

Early binding of parameters (2)

- Binds to the most specific matching parameter list

```java
public static void whatAmI(EnergyConsumer ec) {
    System.out.println("an EnergyConsumer");
}

public static void whatAmI(TV tv) {
    System.out.println("a TV");
}

public static void whatAmI(IsSellableItem isi) {
    System.out.println("an item");
}
```

```java
public static void main(String[] a) {
    TV tv = new TV(32);
    whatAmI(tv);
    whatAmI((IsSellableItem) tv);
    whatAmI((EnergyConsumer) tv);
}
```

```
Compile & Run Lecture3:

a TV
an item
an EnergyConsumer

Success
```

Early binding of parameters (3)

- Binds to the most specific matching parameter list

```java
public static void whoAmI(EnergyConsumer ec) {
    System.out.println("an EnergyConsumer");
}


public static void whoAmI(IsSellableItem isi) {
    System.out.println("an item");
}


public static void whoAmI(PaymentMethod pm) {
    System.out.println("a PaymentMethod");
}
```

```java
public static void main(String[] a) {
    EnergyConsumer tv = new TV(32);
    CreditCard c = new CreditCard(..);
    whoAmI(tv);
    whoAmI(c);
}
```

```
Compile & Run Lecture3:

an EnergyConsumer
a PaymentMethod

Success
```

Early binding of parameters (4)

- Binds to the most specific matching parameter list

```
public static void whoAmI(EnergyConsumer ec) {
    System.out.println("an EnergyConsumer");
}


public static void whoAmI(IsSellableItem isi) {
    System.out.println("an item");
}


public static void whoAmI(PaymentMethod pm) {
    System.out.println("a PaymentMethod");
}
```

```
public static void main(String[] a) {
    TV tv = new TV(32);
    CreditCard c = new CreditCard(..);
    whoAmI(tv);
    whoAmI(c);
}
```

```
Compile Lecture3:

Ambiguous call to whoAmI

Error
```

Mimicking late binding for parameters

- Late binding of parameters is called double dispatch
- Can be supported indirectly via a Dynamic Dispatcher trick
- We consider an example. Complicated!
- Details are discussed in the lecture on design pattterns (Lecture 10)

Dynamic Dispatching Example (Setup)

```java
public interface Dispatcher {
    default void run(DDSupporter thing) { thing.getState(this); }

    void accept(TV tv);
    void accept(Heater h);
}
public interface DDSupporter {
    void getState(Dispatcher d);
}
public class TV implements IsSellableItem, EnergyConsumer {
    public void getState(Dispatcher d) { d.accept(this); }
}
public interface EnergyConsumer extends DDSupporter {}
```

The extends keyword is explained next lecture. Read implements

Using the Dynamic Dispatcher

```java
public class WhoAmI implements Dispatcher {
    @Override
    public void accept(TV tv) { System.out.println("a TV"); }

    @Override
    public void accept(Heater h) {
        System.out.println("a heater");
    }
}
```

```java
public static void main(String[] args) {
    EnergyConsumer ec = new TV(30);
    WhoAmI wai = new WhoAmI();
    wai.run(ec);
    whatAmI(ec);
}
```

```
Compile & Run Lecture3:

a TV
an EnergyConsumer

Success
```

Polymorphism (so far)

- Objects have multiple types
  - at compile time
  - at runtime
- Method invocation:
  - Matches on this object at runtime
  - Matches on parameters at compile time
  - Matching parameters at runtime via dynamic dispatching

Which method to call

- There is more to say than just about binding!
- “Overloading versus overriding versus hiding” next week.

Goal Today

After today's lecture, you should be able to explain

- Default methods
- Polymorphism, early and late binding
- Composition
- Creating objects without explicit types / static factory methods
- Markers

Creating big classes/types

- Separation of concerns: One interface per type of behavior
- Challenge: groups of types with lots of variability in their behavior
  - Phones with different sensors
  - Lecture halls with different projectors
  - …
- We don’t want to create too many different classes

Example: Location on CellPhones

- We want to model a `CellPhone`.
- We have three different types of CellPhones, with different sensors for location.
- Idea: Make an interface for `CellPhone`, then subtypes for `CellPhoneWithGPS`, `CellPhoneWithCellular`, `CellPhoneWithWIFI`
- Also cell phones with 4G or 5G and (no) Wifi
- Phones can have or not have an AudioJack, a Pencil, …
- Exponentially many combinations; not a good idea to create a separate class for each.

Encapsulation (Round three)

1. Change the state of an object only via its behavior.
2. Hide how you store the state, so you can change this later.
3. Composition: Identify what varies & separate it from what stays the same

Composition is powerful

- Instead of creating subtypes, create a class that contains multiple fields.
- More flexible than subtypes, as the fields can dynamically change
- However, whether a combination is meaningful must be checked at runtime.

Cellphone Example with Composition

```
public class CellPhone {
    private PositionProvider posProvider;

    CellPhone(PositionProvider posProvider) {
        this.posProvider = posProvider;
    }

    public void setPositionProvider(PositionProvider newPosProvider) {
        this.posProvider = newPosProvider;
    }

    public Position getPosition() {
        return posProvider.getPosition();
    }
}
```

Cellphone Example with Composition

```
public static void main(String[] args) {
    PositionProvider pp = new GpsPositionProvider()
    CellPhone cp = new CellPhone(pp);
    cp.getPosition();
    cp.setPositionProvider(new CellularPositionProvider());
    cp.getPosition();
}
```

Goal Today

After today's lecture, you should be able to explain

- Default methods
- Polymorphism, early and late binding
- Composition
- Creating objects without explicit types / static factory methods
- Markers

Creating Objects

- In the example above, we still must know about the concrete classes
- In fact, creating objects requires new and this requires a concrete type
- However, we can hide this logic from users

“Named Constructors” or Static Factories

- String.valueOf(int) is easier to understand than String(int)
- Date(int,int,int) or Date.newYearMonthDay()
- Date.newYearMonthDay(int,int,int) and Date.newDayMonthYear(int,int,int)

- Static factories can return subtypes!

PositionProvider with Static Factory (1)

```java
public interface PositionProvider {
    public Position getPosition();
    public static PositionProvider makePositionProvider(String type) {
        switch(type) {
            case "GPS":
                return new GPSPositionProvider();
            case "Cellular":
                return new CellularPositionProvider();
            default:
                return null;
        }
    }
}
```

PositionProvider with Static Factory (2)

```java
public static void main(String[] args) {
// Consider moving the factory into the CellPhone
    PositionProvider pp = PositionProvider.makePositionProvider("GPS");
    CellPhone cp = new CellPhone(pp);
    cp.getPosition();
    cp.setPositionProvider(PositionProvider.makePositionProvider("Cellular"));
    cp.getPosition();
}
```

PositionProvider with Static Factory (3)

```
public static void main(String[] args) {
    CellPhone cp = new CellPhone(“GPS”);
    cp.getPosition();
    cp.setPositionProviderFromString(“Cellular”);
    cp.getPosition();
}


public class CellPhone {
    private PositionProvider posProvider;

    CellPhone(String str) { this.posProvider = PositionProvider.makePositionProvider(str);
    public void setPositionProvider(String newPosProvider) {
        this.posProvider = PositionProvider.makePositionProvider(newPosProvider);
    }
…
}
```

Looking Back (Composition/Static Factories)

- CellPhone can be extended to any new PositionProvider
- User of CellPhone does not need to know how this is implemented

Limitations?

- The static factory method must depend on concrete types.
- The static factory is stateless
- The static factory creates only a single object
- Other factory ‘patterns’ such as factory method and abstract factory
- Main idea: Capture Factory in a separate class

Outlook beyond Static Factories

- Builders
- Dynamic Factories

Builders

- We may not want to have setters for position providers in the cellphone (consider: we cannot dynamically add gps to a phone if the sensor is not available)
- (Static) constructors are not very flexible. We want to add position providers and other sensors flexibly

- Simple but effective solution: Create a builder class that has the setters and a public `build` method that creates a cell phone

(short) Dynamic Factory Method

```
public interface PositionProviderFactory {
    PositionProvider create();
}
```

```
public class GpsPositionProviderFactory implements PositionProviderFactory
{
    public GPSPositionProvider create() {
        return new GPSPositionProvider();
    }
}
```

```
public static void main(String[] args) {
    PositionProviderFactory ppfactory = new GpsPositionProviderFactory();
    CellPhone cp = new CellPhone(ppfactory.create());
}
```

Goal Today

After today's lecture, you should be able to explain

- Default methods
- Polymorphism, early and late binding
- Composition
- Creating objects without explicit types / static factory methods
- Markers

Marker Interface

- Interfaces without any behavior
- Useful to “mark” special types.
- Can be helpful if a method should only be called on special classes

- Java examples: Cloneable, Serializable.

- Own example: HasLongToString

HasLongString example

```
public interface HasLongString { }
public class ShoppingCart implements HasLongString  { … }
public static void printShortString(HasLongString obj) {
    System.out.println(obj.toString().substring(0,8) + "...");
}

public static void printShortString(Object obj) {
    System.out.println(obj);
}
```

Object is explained next lecture. Every class `implements` Object

```
public static void main(String[] args) {
    ShoppingCart cart = new ShoppingCart()
    TV t1 = new TV(32);
    cart.add(t1);
    printShortString(t1);
    printShortString(cart);
}
```

```
Compile & Run Lecture3:
TV (32”)
Cart w…

Success
```

Interfaces

- Interfaces define types
- Polymorphism, early and late binding, which methods are executed
- Do not use implement interfaces everywhere, use, e.g., Composition
- Static factory methods prevent explicit implementations

Next lecture: 17/2

Inheritance

More method selection

the Object class

Exceptions in Java

Summary of OO fundamentals

Tutorial on Tuesday: Examples with Interfaces