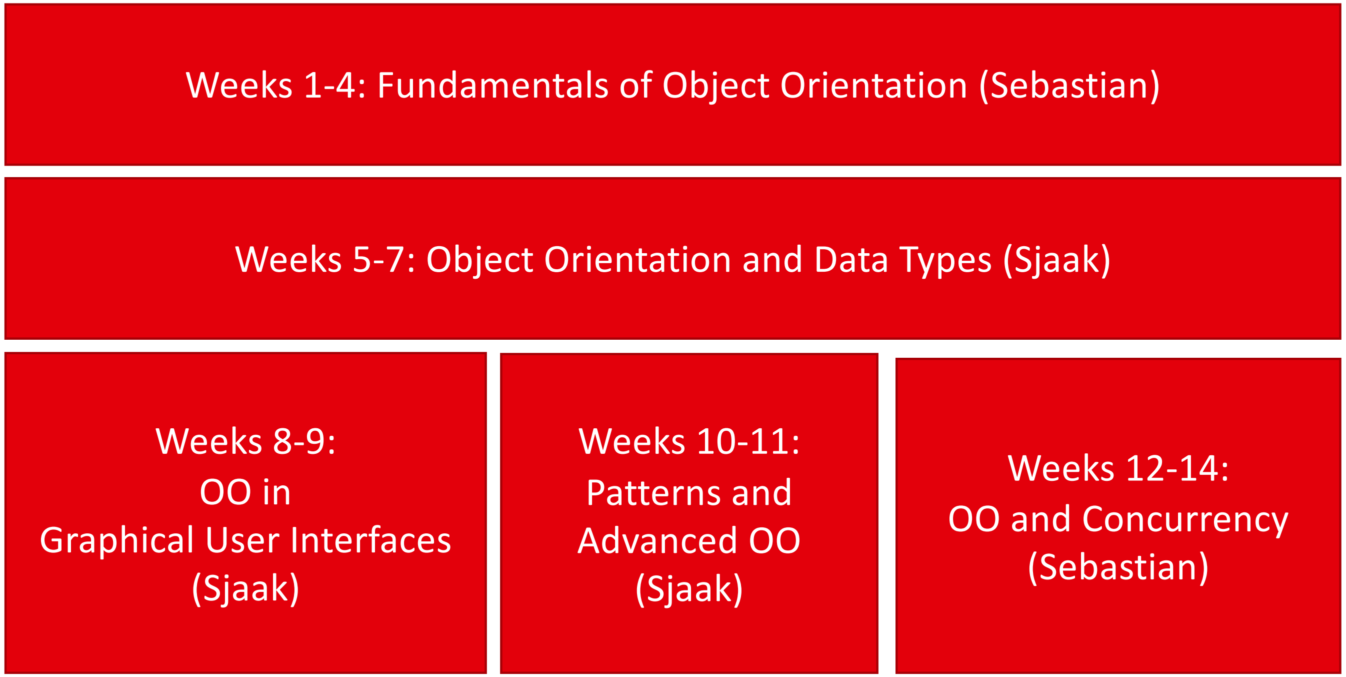
Object-Oriented Programming

Lecture 2: The Memory Model,
Interfaces (part 1)

Sebastian Junges

Course Structure



Weeks 1-4: Fundamentals of Object Orientation (Sebastian)

Weeks 5-7: Object Orientation and Data Types (Sjaak)

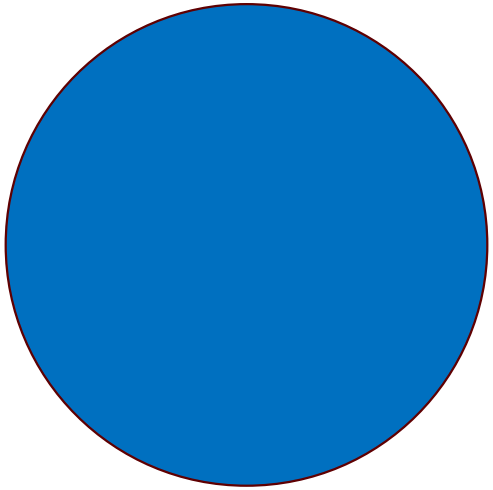
Weeks 8-9:
OO in
Graphical User Interfaces
(Sjaak)

Weeks 10-11:
Patterns and
Advanced OO
(Sjaak)

Weeks 12-14:
OO and Concurrency
(Sebastian)

Goal Today

- Explain the elementary consequences of Java’s memory model
- Discuss Seperation of Concerns
- Intro to Interfaces



Example: Circle

```
public class Circle {

    private int radius;

    public Circle(int circleradius) {
        radius = circleradius;
    }

    public int getRadius() { return radius; }

    public void setRadius(int newRadius) { radius = newRadius; }

}
```

this object

Don’t do this

A method is called on an object.

Example: in class `Circle`:

```
void setRadius (int newRadius)
```

can be simulated by

```
static void staticSetRadius (Circle c, int newRadius)
```

`c1.setRadius(5);` becomes `Circle.staticSetRadius(c1,5);`

Example: Circle

```
public class Circle {

    private int radius;

    public Circle(int circleradius) {
        this.radius = circleradius;
    }

    public int getRadius() { return this.radius; }

    public void setRadius(int newRadius) { this.radius = newRadius; }

}
```

Example: Circle

```
public class Circle {

    private int radius;

    public Circle(int circleradius) {
        radius = circleradius;
    }

    public int getRadius() { return radius; }

    public void setRadius(int newRadius) { radius = newRadius; }

}
```

Inside a class, the this object is passed implicitly.

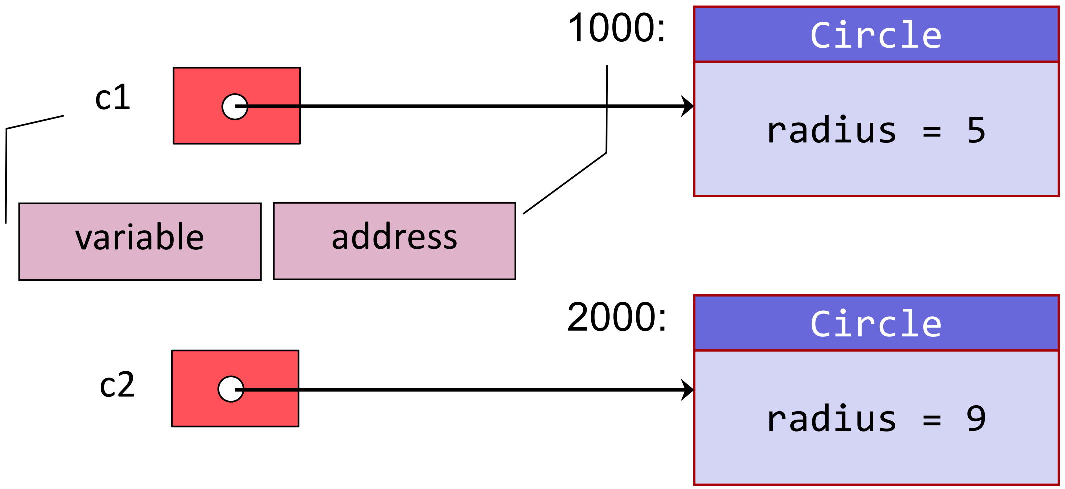
Java Memory Model Basics

How do we refer to these objects and where do we store them?

References and addresses

Variable for Reference Type
refers to an object

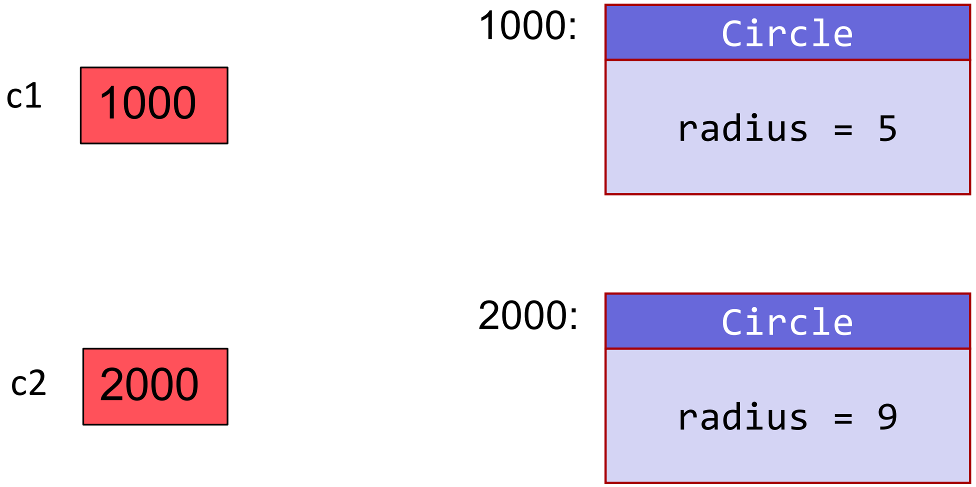
```
public static void main(String[] args) {
    Circle c1 = new Circle(5);
    Circle c2 = new Circle(9);
}
```



References and addresses

Variable contains an address

```
public static void main(String[] args) {
    Circle c1 = new Circle(5);
    Circle c2 = new Circle(9);
}
```



c1 1000

c2 2000

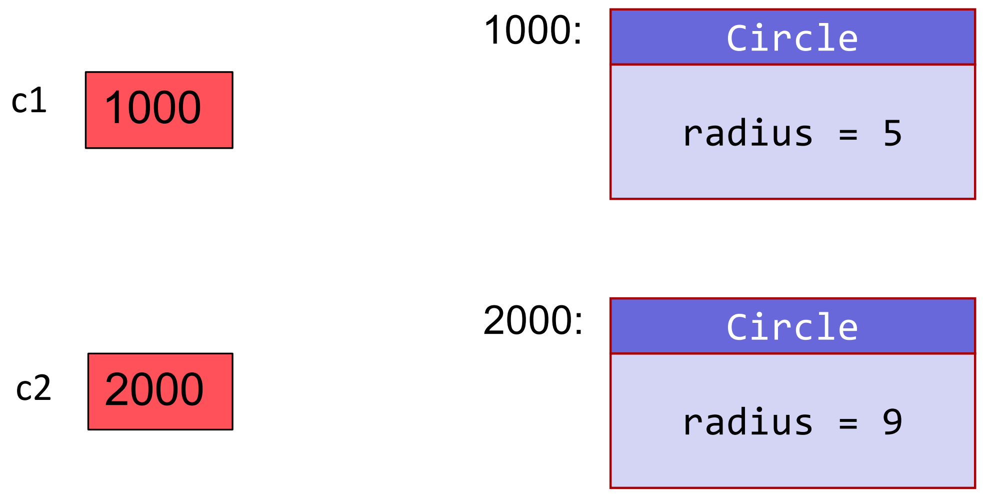
1000: Circle radius = 5

2000: Circle radius = 9

References and addresses

Variable contains an address

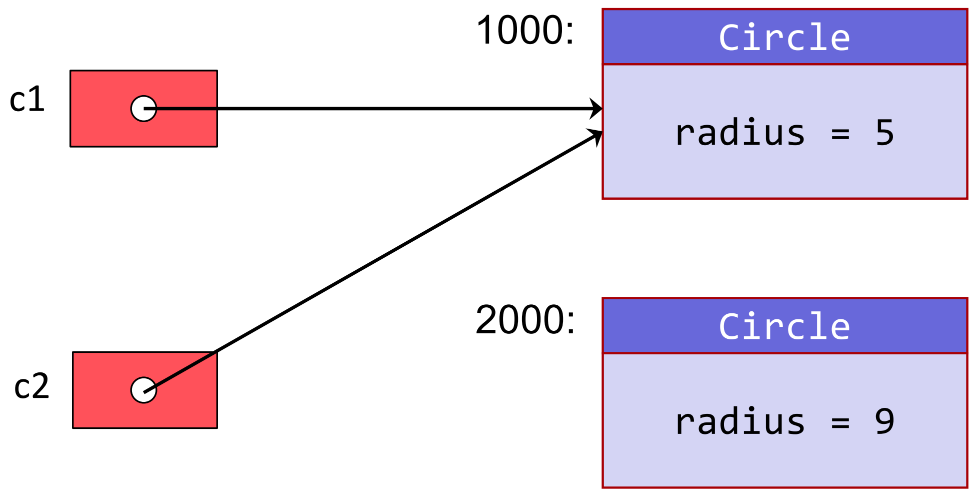
```java
public static void main(String[] args) {
    Circle c1 = new Circle(5);
    Circle c2 = new Circle(9);
}
```



Memory management: Aliasing

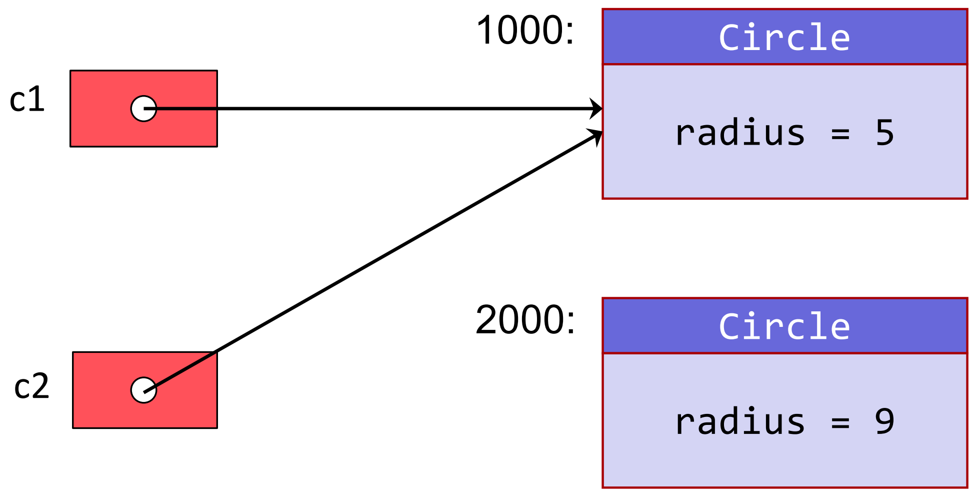
Different names for the same object, for example after the assignment

```
public static void main(String[] args) {
    Circle c1 = new Circle(5);
    Circle c2 = new Circle(9);
    c2 = c1;
}
```



Memory management: Garbage collection

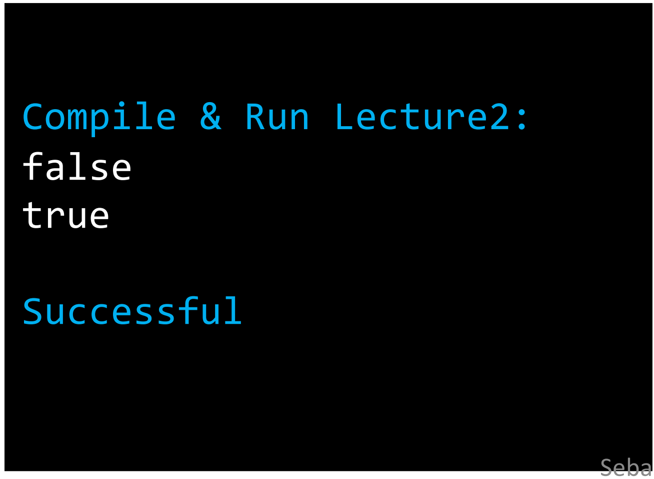
reclaiming space of unused objects



Java Memory Model Basics
Some elementary consequences

Comparing reference types

```java
public static void main(String[] args) {
    Circle c1 = new Circle(5);
    Circle c2 = new Circle(5);
    System.out.println(c1 == c2);
    c1 = c2;
    System.out.println(c1 == c2);
}
```



```
Compile & Run Lecture2:
false
true

Successful
```

Comparing reference types

```java
public static void main(String[] args) {
    Circle c1 = new Circle(5);
    Circle c2 = new Circle(9);
    System.out.println(c1.equals(c2));
    c2.setRadius(5);
    System.out.println(c1.equals(c2));
}

public class Circle {
    private int radius;

    public boolean equals(Circle other) {
        // In future lectures, we will improve this method
        return (this.radius == other.radius);
    }
}
```

```
Compile & Run Lecture2:
false
true

Successful
```

Primitive vs. Reference Types

```java
public static void mainPrim() {
        int x = 10;
        int y = 20;
        System.out.println( x );
        System.out.println( y );
        y = x;
        x = 50;
        System.out.println( x );
        System.out.println( y );
}
```

```
Compile & Run Lecture2:
10
20
50
10

Successful
```

```java
public static void mainPrint() {
    Circle c1 = new Circle(5);
    Circle c2 = new Circle(9);
    System.out.println(c1);
    System.out.println(c2);
    c2 = c1;
    c1.setRadius(2);
    System.out.println(c1);
    System.out.println(c2);
}
```

```
Compile & Run Lecture2:
Circle w radius 5
Circle w radius 9
Circle w radius 2
Circle w radius 2

Successful
```

Unitialized reference variables

```java
public static void main(…) {
    Circle c1 = new Circle(5);
    Circle c2;
    c2.setRadius(7);
}
```

```
Compile ~~& Run Lecture2~~:
variable c2 might not have been
initialized

Error
```

```java
public static void main(…) {
    Circle c2 = null;
    c2.setRadius(7);
}
```

```
Compile & Run Lecture2:
Cannot invoke
“lecture2.Circle.setRadius(int)”
because “c2” is null

Error
```

```java
public static void main(…) {
    Circle[] cs = new Circle[3];
    cs[0].setRadius(7);
}
```

```
Compile & Run Lecture2:
Cannot invoke
“lecture2.Circle.setRadius(int)”
because “cs[0]” is null

Error
```

Today

- Java’s memory model
- Seperation of Concerns
- Interfaces

To create an effective code base:

- Think about the need to change your code!
- Allow to reuse as much of your code!

Encapsulation (second round)

- (Round 1): Change the state of an object only via its behavior.
- Hide how you store the state, so you can change this later.

Changing the **`Door`** class

- State consists of Booleans `isClosed` and `isLocked`
- Assume we want to support putting a door ajar
- `isClosed` becomes a double `howMuchClosed`
- Change all methods to use `howMuchClosed` instead of `isClosed`
- The users of the **`Door`** class are not affected by this change!

Methods! (1, Last week)

```
public class Door {
    private boolean isClosed;
    private boolean isLocked;
    private String material;


    …

    public void open() {
        if (!isLocked) {
            isClosed = false;
        }
    }

    public boolean isOpen() {
        return !isClosed;
    }
}
```

Methods! (2, Changed)

```
public class Door {
    private double howMuchClosed;
    private boolean isLocked;
    private String material;


    …

    public void open() {
        if (!isLocked) {
            howMuchClosed = 0.0;
        }
    }

    public boolean isOpen() {
        return howMuchClosed < 0.0;
    }


    public boolean isWideOpen() {

         return howMuchClosed == 0.0;

    }
}
```

New Example: `Date` (on a calendar)

- State: Date
- Behavior:
  - Get the next week
  - Export as String in (some standard) format
  - …

Encapsulation Example

```java
public class Date
{
    private int year;
    private int month;
    private int day;
}
```

```java
public class Date
{
    private int julian;
}
```

Number of days since
Jan. 1, 4713 BCE

E.g.:
2021, 2, 2 is
2 February 2021

E.g:
2.459.248 is
2 February 2021

Encapsulation Idea

- (Meaning of) methods is independent of the implementation
- User does mostly not need to know
  (performance typically not that important)
- Hiding details from the user is an argument to not provide a setter (mutator) for every field (example below)

Mutator for multiple attributes (1)

```java
public class Date {
    private int day, month, year;

    public boolean setDay(int newDay) {
        // Exceptions are introduced later.
        if (isValid(newDay, month, year)) {
            day = newDay; return true;
        }
        return false;
    }

    private static boolean isValid(int day, int month, int year) {
        if (day > 31) { return false; }
        if (month == 4 || month == 6 || month == 9 || month == 11) {
            return day != 31;
        }
        // ....
        return true;
    }
}
```

Mutator for multiple attributes (2)

```
public static void main(String[] args) {
    Date d1 = new Date(31,10,2022);
    d1.setMonth(11);
    d1.setDay(30);
    // First month then day, problem!
    Date d2 = new Date(30, 11, 2022);
    d2.setDay(31);
    d2.setMonth(10);
    // First day then month, problem!
}
```

- Better: `setDate(int day, int month, int year)`

Extending `Date` to `Meeting`

- We want to add meeting information
  - `addParticipant`(…)
  - `addRoom`(…)
- We can add all of this in the class for `Date`. Don’t do this!
- We now want to use `Date` in the `Student` class for birthdays
  - Birthdays cannot have a room…

Separation of Concerns

If a class does one thing and only one thing, it is:

- easier to understand
- easier to design
- easier to implement
- easier to test
- easier to debug
- easier to reuse

Separation of Concerns and Encapsulation

If the user does not need to know the internals

- easier to use for the user
- easier to optimize for the programmer
- easier to extend for the programmer

Side remark (for homework!)

- Input/output is a separate concern from data storage
- -> Do not put I/O into your data classes

Problem: Tight coupling

- Say we are implementing the class Meeting:
  We must pick a fixed Date class for day,

- but we do not care about its implementation.

- Solution: Interfaces

Today

- Java’s memory model
- Seperation of Concerns
- Interfaces

“Program to an interface, not an implementation”

Introduction to interfaces

- In General, an interface is the place at which independent and often unrelated systems meet and act on or communicate with each other [*Merriam-Webster*]
  - A language is an interface between two people.
  - A remote control is an interface between you and a television.
- In Computing, an interface is a shared boundary across which two or more components of a computer system exchange information.
- In object oriented programming, an interface is a common means for unrelated objects to communicate with each other.

Interfaces as Abstractions

Users do not care about the implementation, only about the behavior

```java
public interface Date {
    public int getYear();
    public int getMonth();
    public int getDay();
    public Date nextWeek();
    public Date nextYear();
}
```

```java
public class Meeting {
    //
    public Meeting(Date date,
                   String title) {
        //...
    }
}
```

Java interfaces

A Java `interface` contains:

- method specifications (no implementation), called abstract methods
- default methods
- constant definitions.

A Java `interface` does not contain:

- constructors,
- instance variables.

Class Implementing an Interface

```
public class StandardDate implements Date {
    private int day, month, year;

    @Override
    public int getYear() { return this.year; }

    @Override
    public int getMonth() { return this.month;  }

    @Override
    public int getDay() { return this.day; }

    @Override
    public Date nextYear() {
        if ((day == 29) && (month == 2)) {
            return new StandardDate(1, 3, year+1); // Or throw an exception
        }
        return new StandardDate(day, month, year+1);
    }
}
```

Using Interfaces

```
public class JulianDate implements Date {
    private int days;

    @Override
    public int getYear() {
        // ...
    }
}

public static void main(String[] args) {
    Date d1 = new StandardDate(31,10,2022);
    Date d2 = new JulianDate(31,10,2022);
    Meeting m1 = new Meeting(d1, "Lecture");
    Meeting m2 = new Meeting(d2, "Tutorial");
    Meeting m3 = new Meeting(d1.nextYear(), "Another lecture");
}
```

Interfaces as Contracts

Goal: minimizing dependencies.

Interfaces separate the
concern of the implementation (server) from the *concern* of the user (client)

Allows client and server to be
developed, implemented, tested, optimized, used, understood, and modified independently.

Interface specifies a contract:

- *maximum* functionality a client can use
- *minimum* functionality an server has to provide

Example: Payment (1)

```java
public interface PaymentMethod {
    boolean pay(double amount);
}
```

PaymentMethod can represent completely different types of objects, as long as we can pay with them…

Example: Payment (2)

```java
public class Cash implements PaymentMethod {
    private int fiftyEuro, twentyEuro, tenEuro, fiveEuro;

    public Cash(int fifty, int twenty, int ten, int five) {
        fiftyEuro = fifty;
         // ...
    }

    public int total() {
        return fiftyEuro * 50 + twentyEuro * 20 + tenEuro * 10 + fiveEuro * 5;
    }

    @Override
    public boolean pay(double amount) {
        if (total() < amount) { return false; }
        if (amount < 5 && fiveEuro > 0) {
            fiveEuro -= 1; // NO CHANGE...
        }
        // ...
        return true;
    }
}
```

Example: Payment (3)

```
public class CreditCard implements PaymentMethod{
    private final int number;
    private final int cvc;
    private final String name;
    private double remainingCredit;

    public CreditCard(String name, int number, int cvc, double
initialCredit) {
        this.name = name; this.number = number; this.cvc = cvc;
        remainingCredit = initialCredit;
    }

    @Override
    public boolean pay(double amount) {
        if (remainingCredit > amount) {
            remainingCredit -= amount;  return true;
        }
        return false;
```

Loose Coupling

Encapsulation and separation of concerns require loose coupling of different code parts.

Interfaces are the key ingredient to provide this loose coupling

Interfaces (summary)

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

- Signature (return type, arguments) are enforced by the compiler
- Semantic meaning is not!

Example: ShoppingCart

- Online store:
  - a user buys things by adding them to a shopping cart,
  - ordering everything that is in the cart.

- Responsibilities for the ShoppingCart class:
  - Know which things are in the shopping cart.
  - Allow to add things
  - Allow ordering everything in the cart.

- What types can the ShoppingCart hold?

Example: Sellable Items

```
public interface IsSellableItem {
    public double getPrice();
}
```

```
public class TV implements IsSellableItem {
    private int size;

    public TV(int size) {
        this.size = size;
    }

    public int getSize() { return this.size; }

    @Override
    public double getPrice() { return 500; }
}
```

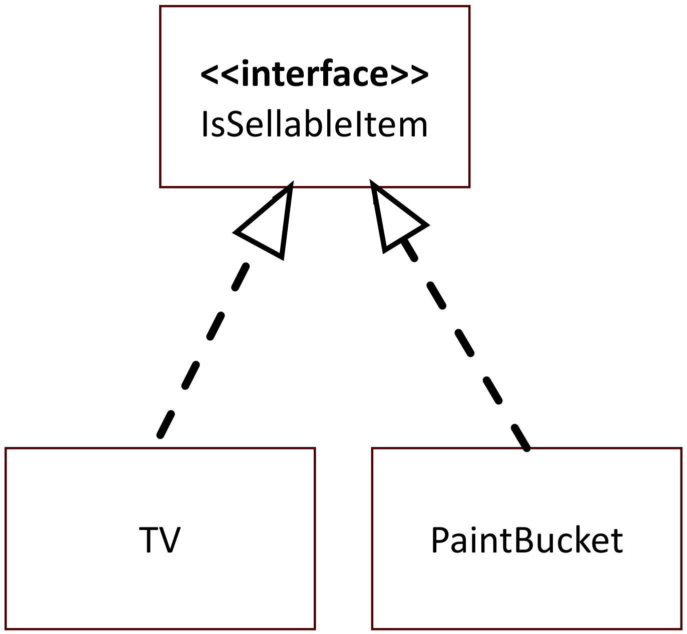
Shopping Cart (1)

```
public class ShoppingCart {
    private IsSellableItem[] items;
    private int numberItems = 0;
    public static final int CAPACITY = 30;

    public ShoppingCart() {
        items = new IsSellableItem[CAPACITY];
    }

    public void add(IsSellableItem newItem) {
        if (numberItems < CAPACITY) {
            items[numberItems] = newItem;
            numberItems++;
        }
    }
    // To be continued
}
```

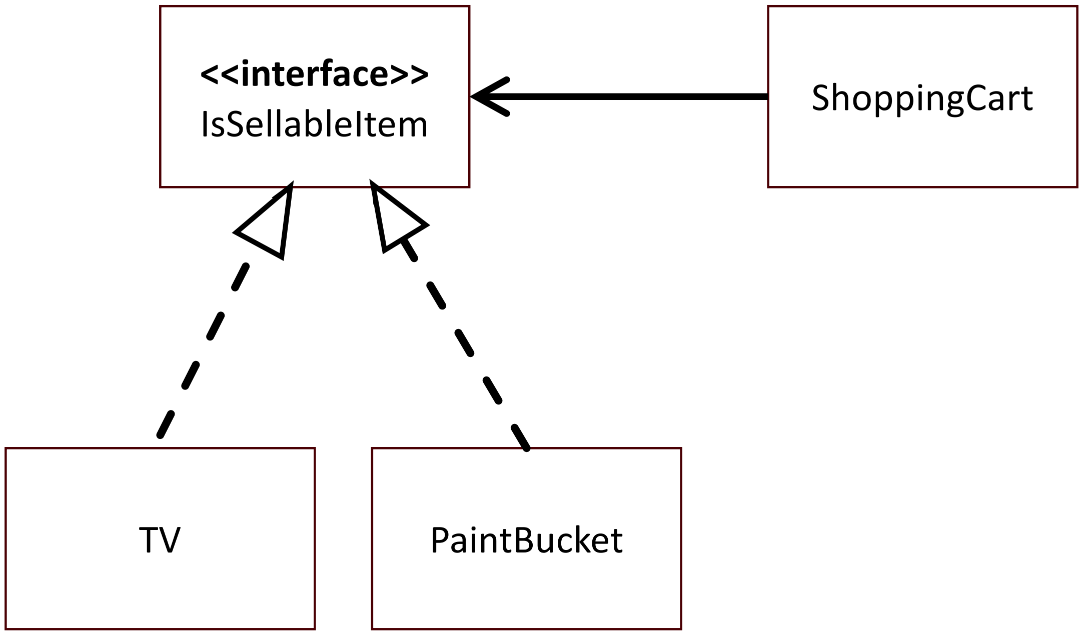
Shopping Cart (2)

```java
public class ShoppingCart {
     // ...
    private double value() {
        double sum = 0;
        for (int i = 0; i < numberItems; i++) {
            sum += items[i].getPrice();
        }
        return sum;
    }

    public boolean checkout(PaymentMethod payment) {
        return payment.pay(value());
    }
}
```

Class Diagram



Class Diagram

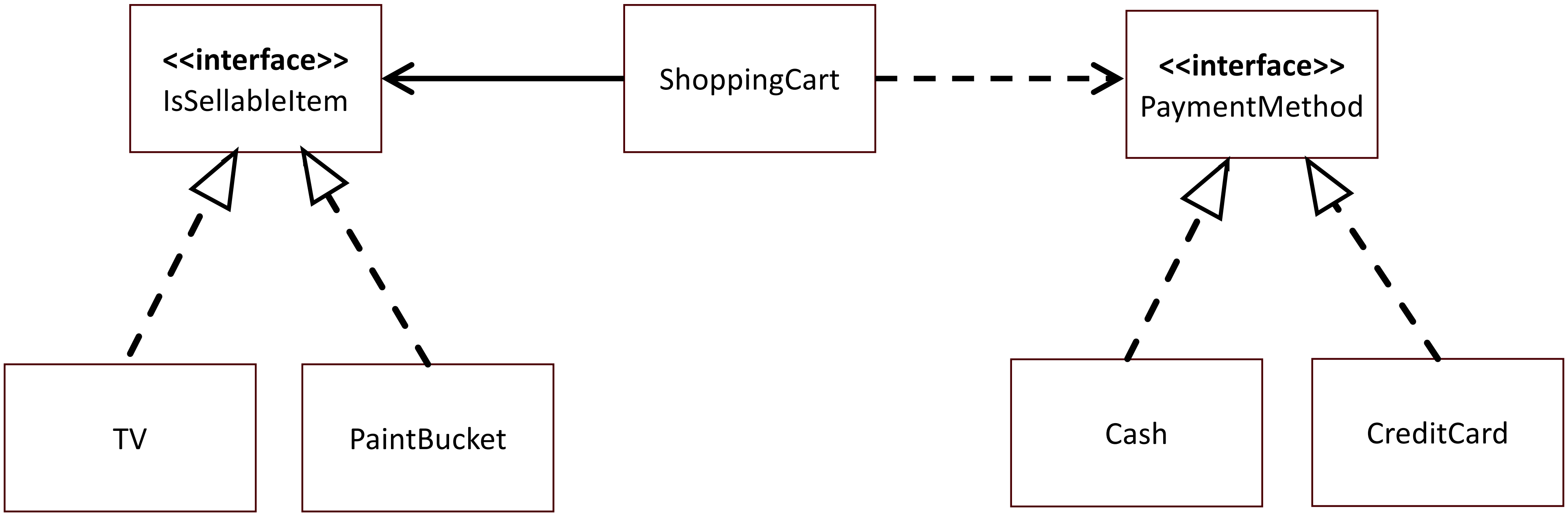


Example: Payment (1)

```java
public interface PaymentMethod {
    boolean pay(double amount);
}
```

PaymentMethod can represent completely different types of objects, as long as we can pay with them…

Class Diagram



<<interface>>
IsSellableItem

ShoppingCart

<<interface>>
PaymentMethod

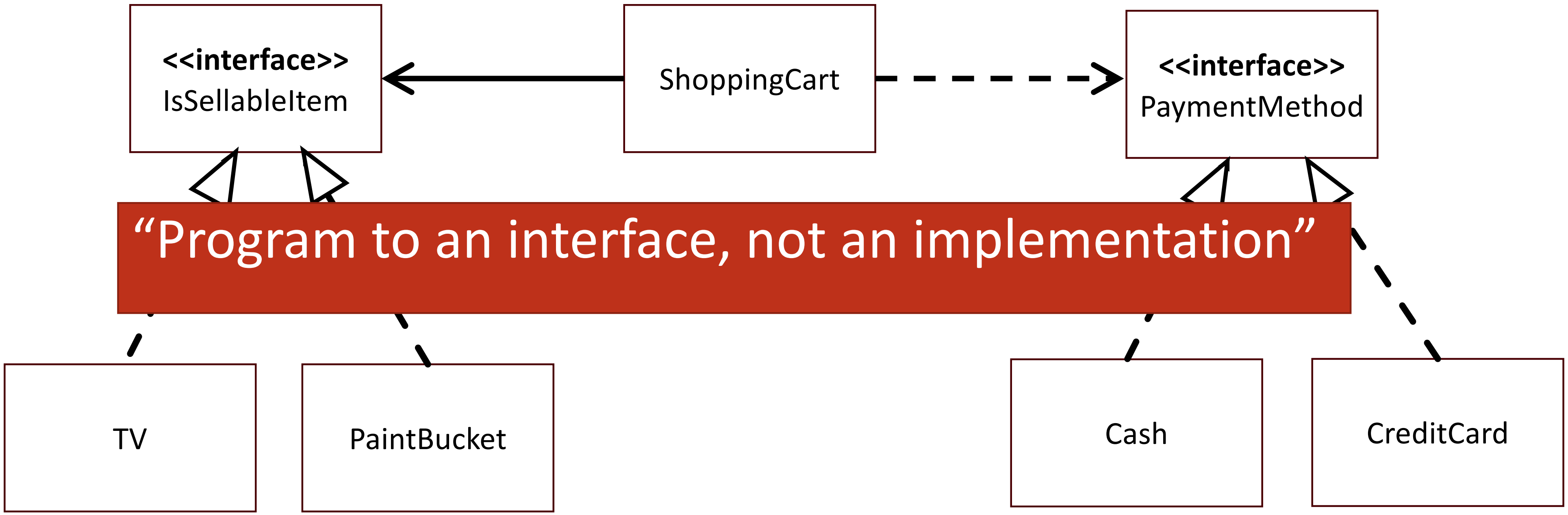
TV

PaintBucket

Cash

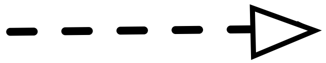
CreditCard

Class Diagram

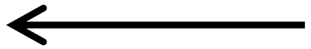


“Program to an interface, not an implementation”

(UML) Class Diagrams

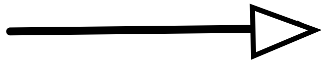


implements

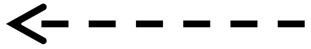


association (for fields)

next lecture

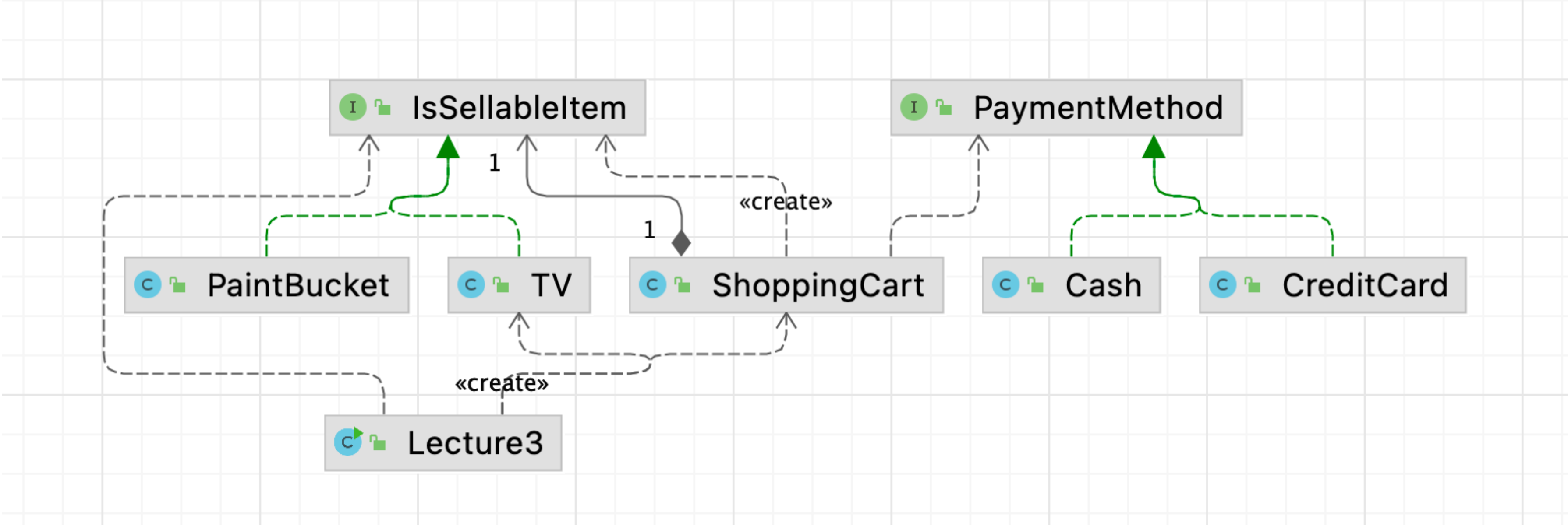


extends



dependency (for args/return)

Auto-generated Class Diagrams



Class diagrams help show dependencies between code

What is the type of an object?

- an object can be Cash, which is a PaymentMethod
- an object can be a TV, which is a sellable item
- ….

Interfaces and types

An interface defines a (reference) type.

It can be used as the type of a field, a local variable, or a parameter.

- `TV` and `PaintBucket` are subtypes of `IsSellableItem`
- `Cash` and `CreditCard` are subtypes of `PaymentMethod`
- `IsSellableItem` is a supertype of `TV` and `PaintBucket`
- `PaymentMethod` is a supertype of `Cash` and `CreditCard`

Types and Subtypes

Subtype rule:

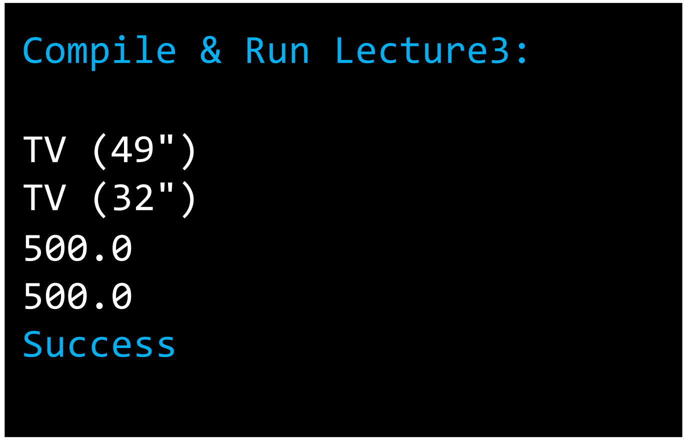
If a method or variable requires a reference of type A,
then a value of any subtype of A may be provided.

Thus for adding items to the shopping cart:

- It is specified to accept `IsSellableItems`;
- It can be invoked with any variable of type `IsSellableItem`, or any variable with that subtype.

Static vs Dynamic Types

```java
public static void main(String[] args) {
    ShoppingCart cart = new ShoppingCart();
    TV tv1 = new TV(49);
    IsSellableItem tv2 = new TV(32);
    System.out.println(tv1);
    System.out.println(tv2);
    System.out.println(tv1.getPrice());
    System.out.println(tv2.getPrice());
    cart.add(tv1);
    cart.add(tv2);
}
```



```
Compile & Run Lecture3:

TV (49")
TV (32")
500.0
500.0
Success
```

Static vs Dynamic Types

```java
public static void main(String[] args) {
    TV tv1 = new TV(49);
    IsSellableItem tv2 = new TV(32);
    System.out.println(tv1.getSize());
    System.out.println(tv2.getSize());
}
```

```
Compile Lecture3:

java: cannot find symbol
  symbol:   method getSize()
  location: variable tv2 of type  lecture3.IsSellableItem
Error
```

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.

Static and Dynamic type of an object

The static type

- 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.

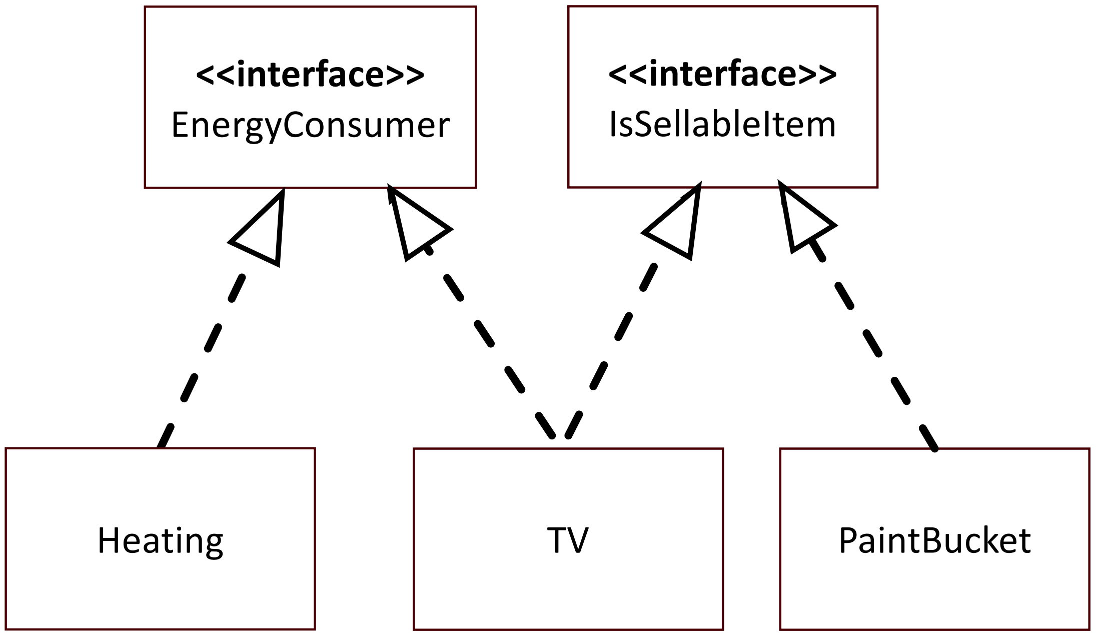
(Object) Polymorphism

- 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

Implementing multiple interfaces



Implementing multiple interfaces

```java
public class TV implements IsSellableItem, EnergyConsumer {
    private int size;

    public TV(int size) { this.size = size; }

    public int getSize() { return this.size; }

    @Override
    public double getPrice() { return 500; }

    @Override
    public int getPower() { return 34; }

}
```

Downcasting (1)

Try to avoid

```
public static void main(String[] args) {
    TV tv1 = new TV(49);
    IsSellableItem tv2 = new TV(32);
    System.out.println(tv1.getSize());
    System.out.println(((TV)tv2).getSize());

}
```

```
public static void main(String[] args) {
    TV tv1 = new TV(49);
    IsSellableItem tv2 = new TV(32);
    System.out.println(tv1.getSize());
    System.out.println(((EnergyConsumer)tv2).getPower());

}
```

Downcasting (2)

Try to avoid

```
public static void main(String[] args) {
    TV tv1 = new TV(49);
    IsSellableItem tv2 = new TV(32);
    System.out.println(tv1.getSize());
    System.out.println(((PaymentMethod)tv2).pay());

}
```

```
Compile & Run Lecture3:

Exception java.lang.ClassCastException:
class lecture3.TV cannot be cast to class lecture3.PaymentMethod
        at Lecture3.main(Lecture3.java:8)

Error
```

instanceof

Try to avoid

```java
public static void main(String[] args) {
    EnergyConsumer h1 = new TV(32);
    EnergyConsumer h2 = new Heater();
    helper(h1);
    helper(h2);
}

public static void helper(EnergyConsumer ec) {
    if (ec instanceof TV) {
        System.out.println("TV");
    } else {
        System.out.println("not a TV");
    }
}
```

```
Compile & Run Lecture3:

TV
not a TV

Success
```

Next lecture: 10/2

Polymorphism & Dynamic binding

Q&A Tuesday: Your questions & examples with interfaces