Programmieren II

Inheritance

Alexander Fraser

fraser@cl.uni-heidelberg.de

(Contains material from T. Bögel, K. Spreyer, S. Ponzetto, M. Hartung)

May 14, 2014

Outline

Outline

Last session

1. Modifiers: `static`, `public`, `private`
2. Overloading methods
3. Constructor chaining

Modifier `static`

- In general, variables of a class are assigned to objects the class instantiates (e.g. the variable `name` of the class `Cat`).
- `static` variables & methods can be accessed directly via the class itself without instantiating an object
- `static` should be used scarcely

Modifiers `public` & `private`

Restricting visibility

- `public` and `private` (and `protected` – later) specify the visibility of class elements (variables, methods)
- `public` elements are visible everywhere
- `private` elements are visible to objects of the class only

Hiding your implementation

Advantages of `private` variable + `public` setter/getter

- Check for inconsistencies
- Implementation independent from interface

Real developers encapsulate!

Without encapsulation...

- Reference variable: `Cat myCat = new Cat();`
- This would be ok: `myCat.height = 27;`
- This would be disastrous: `myCat.height = -1;`

→ we need to protect the cat (and all other objects) from invalid size (and other variable) changes!

Encapsulation

- **Always** choose the most restrictive visibility possible
- To allow changes: use setters and getters
- This process is called **encapsulation**
- Mark instance variables **private**
- Mark getters/setters **public**

Exercise

exercise: Be the Compiler

BE the compiler

Each of the Java files on this page represents a complete source file. Your job is to play compiler and determine whether each of these files will compile. If they won't compile, how would you fix them, and if they do compile, what would be their output?

A

```
class XCopy {

  public static void main(String [] args) {

    int orig = 42;

    XCopy x = new XCopy();

    int y = x.go(orig);

    System.out.println(orig + " " + y);
  }

  int go(int arg) {

    arg = arg * 2;

    return arg;
  }
}
```

B

```
class Clock {
  String time;

  void setTime(String t) {
    time = t;
  }

  void getTime() {
    return time;
  }
}

class ClockTestDrive {
  public static void main(String [] args) {

    Clock c = new Clock();

    c.setTime("1245");
    String tod = c.getTime();
    System.out.println("time: " + tod);
  }
}
```


HF, p. 88

Overloading

- Multiple methods with the same name but different parameters
- Parameters differ in number and/or type

- Often used for constructors:

```
Cat() { ... }
Cat( int age ) { ... }
Cat( int age, boolean isGrumpy ) { ... }
Cat( int age, boolean isGrumpy, double weight ) { ... }
```

Overloading

- Duplicating initialization code in multiple methods should be avoided
- Instead: constructor chaining
- Call simple constructors from more complex ones
- Syntax: `this( ... );`
- Needs to be the first statement in a constructor!

Overloading

```
Cat( int age ) { this.age = age; }

Cat( int age, boolean isGrumpy ) {
   this.age = age;

   this.isGrumpy = isGrumpy;
}

Cat( int age, boolean isGrumpy, double weight ) {
   this.age = age;
   this.isGrumpy = isGrumpy;

   this.weight = weight;
}
```

Overloading

```
Cat( int age ) { this.age = age; }

Cat( int age, boolean isGrumpy ) {
    this.age = age;

    this.isGrumpy = isGrumpy;
}

Cat( int age, boolean isGrumpy, double weight ) {
    this.age = age;
    this.isGrumpy = isGrumpy;

    this.weight = weight;
}
```

Overloading

```
Cat( int age ) { this.age = age; }

Cat( int age, boolean isGrumpy ) {
   this( age );

   this.isGrumpy = isGrumpy;
}

Cat( int age, boolean isGrumpy, double weight ) {
   this.age = age;
   this.isGrumpy = isGrumpy;

   this.weight = weight;
}
```

Overloading

```
Cat( int age ) { this.age = age; }

Cat( int age, boolean isGrumpy ) {
   this( age );

   this.isGrumpy = isGrumpy;
}

Cat( int age, boolean isGrumpy, double weight ) {
   this.age = age;
   this.isGrumpy = isGrumpy;

   this.weight = weight;
}
```

Overloading

```
Cat( int age ) { this.age = age; }

Cat( int age, boolean isGrumpy ) {
   this( age );

   this.isGrumpy = isGrumpy;
}

Cat( int age, boolean isGrumpy, double weight ) {
   this( age, isGrumpy );


   this.weight = weight;
}
```

Outline

The Unified Modeling Language (UML)

UML

- Standardized modeling language
- De-facto industry standard for modeling various aspects of a program/system
- Aim: visual representation of object-oriented software $\rightarrow$ abstraction
- Current version: 2.4.1 (since Aug 2011)
- `www.uml.org`

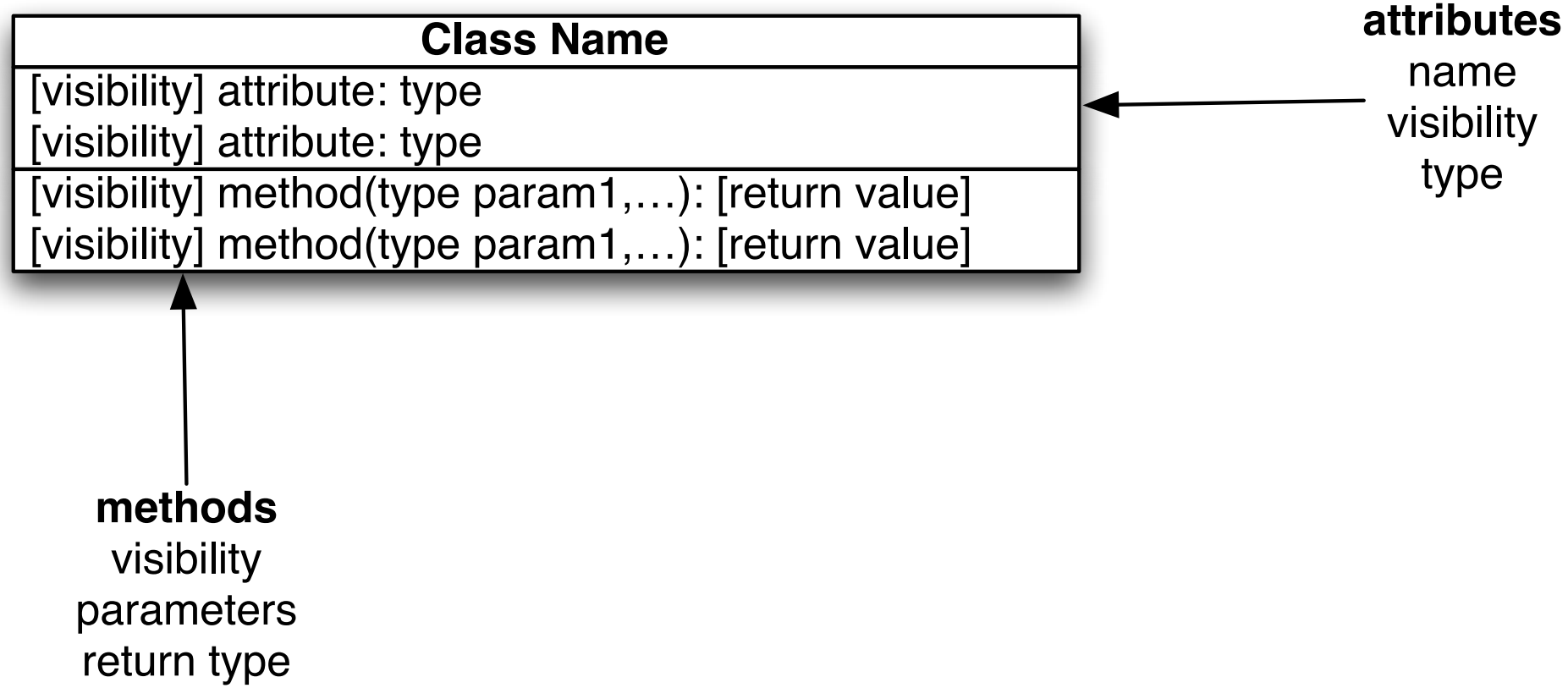
Diagram types in UML

Types of diagrams

- Structural diagrams (static)
  - **Class diagrams**
  - Composite structure diagrams
  - ...
- Behavioral diagrams (dynamic)
  - Sequence diagrams
  - Activity diagrams
  - ...

$\rightarrow$ for now: class diagrams

Class diagram



Class diagram explained

Visiblity

- $+$: public
- $\#$: protected (later today)
- $-$: private

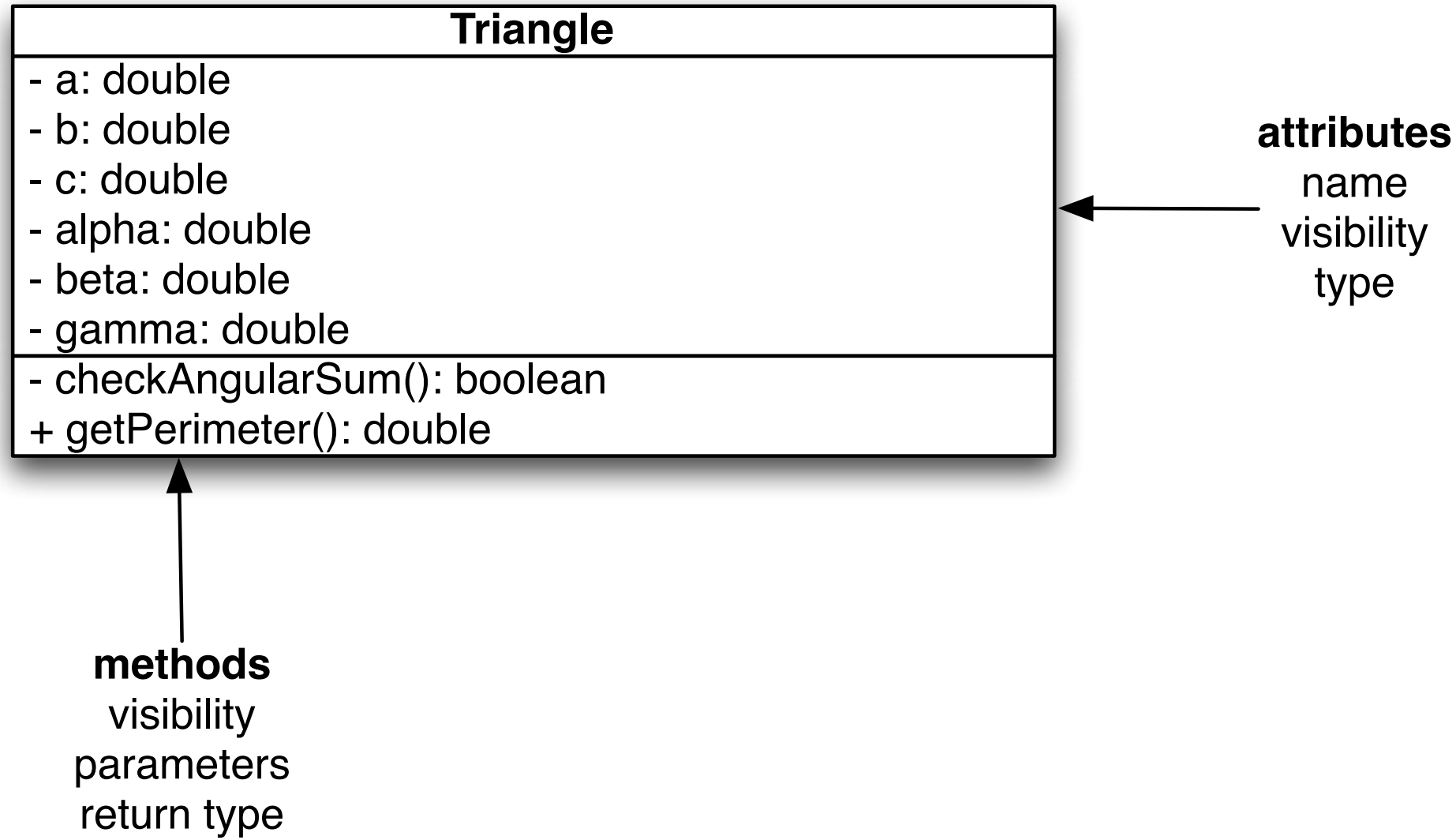
Optionality

- Methods: name and parameter names obligatory
- Variables: name obligatory
- Other information is often not explicitly stated
- Amount of information depends on the **use case** of the diagram (specification vs. general description)

```
class Triangle {
  private double a,b,c, alpha, beta, gamma;

  public setAngles(double a, double b, double g) {
    this.alpha = a;
    this.beta = b;
    this.gamma = g;
    if (! checkAngularSum())
      System.err.println("Invalid angles!");
  }

  private boolean checkAngularSum(){
    return (alpha + beta + gamma == 180.0);
  }

  public double getPerimeter(){
    return a + b + c;
  }
}
```

Triangle.java

Class diagram for `Triangle.java`



UML software

Open source software

- Numerous applications for UML diagrams
- Recommendation: astah (community edition) (`http://astah.net/editions/community`)
- Web-based: yUML (`http://yuml.me/`)

Outline

Class hierarchies

Class hierarchies define a taxonomy of types

- Modeling similarities between classes
  - structural similarities
  - behavioral similarities
- Examples: . . .

Example from Head First Java, pp. 28 ff.

Specification

There will be shapes on a GUI, a square, a circle and a triangle. When the user clicks on the shape, it will rotate clockwise 360° and play an **.aiff** sound specific to the particular shape.

Procedural solution

```
rotate(shapeNum) {
 // make the shape rotate 360 degree
}
playSound(shapeNum) {
 // use shapeNum to lookup which .aiff sound to play
 // play it
}
```

OOP solution I

Class: Square

```
rotate() {
  // code to rotate a square
}
playSound() {
 // code to play the .aiff sound for a square
}
```

Class: Circle

```
rotate() {
  // code to rotate a circle
}
playSound() {
 // code to play the .aiff sound for a circle
}
```

Not so fast: there's been a change! I

Spec addition

There will be an amoeba shape on the screen, with the others. When the user clicks on the amoeba, it will rotate like the others and play a **.hif** sound.

Not so fast: there's been a change! II

Consequences for procedural approach

- `rotate()` still works
- `playSound()` needs to change

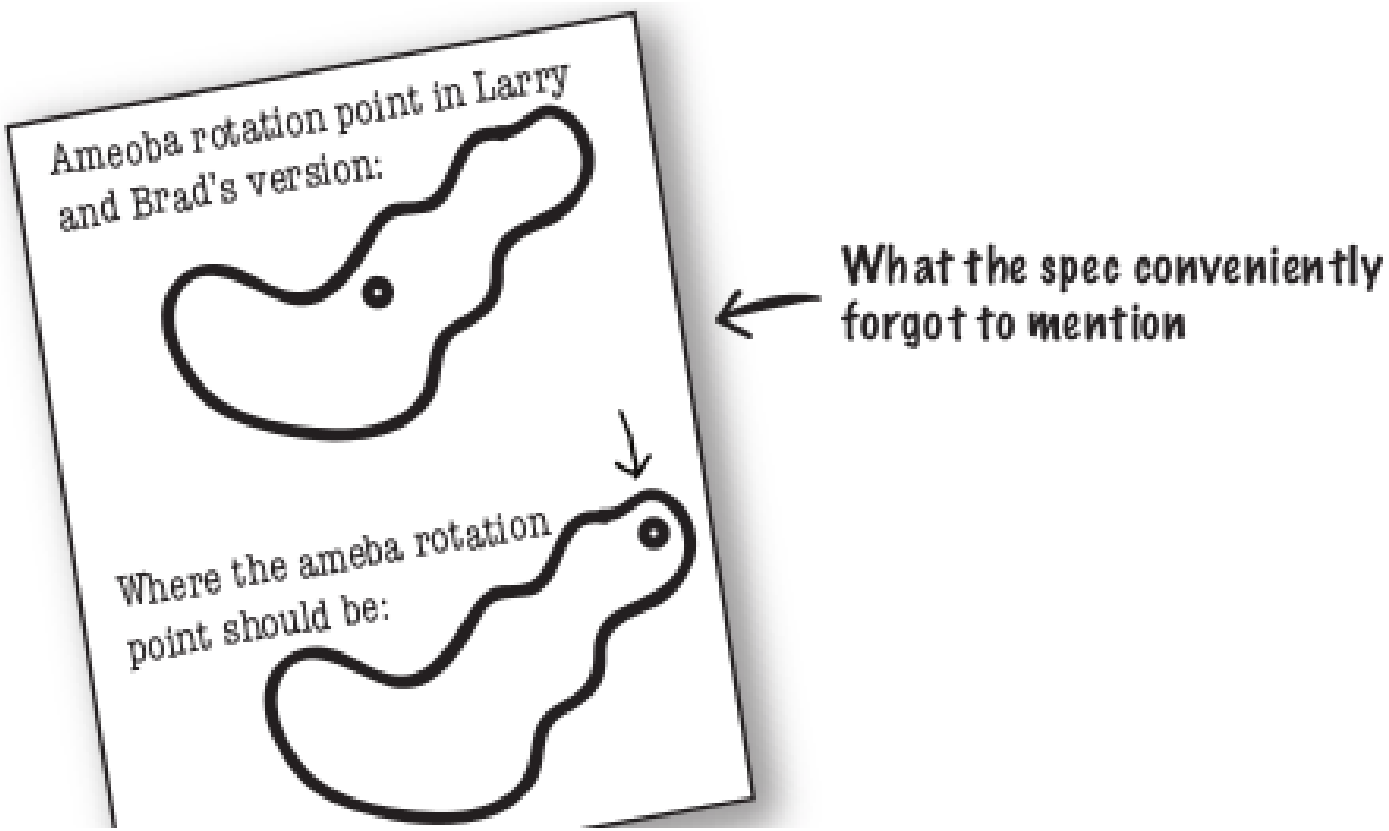
```
playSound(shapeNum) {
// if the shape is not an amoeba,
  // use shapeNum to lookup which
  // AIF sound to play, and play it
// else
  // play amoeba .hif sound
}
```

Changes for OOP approach

- Just create a new class `Amoeba`

Not so fast: there's been a change! III

What the spec forgot to mention...



Not so fast: there's been a change! IV

Consequences for procedural approach

- `rotate()` is getting ugly

```
rotate(shapeNum, xPt, yPt) {
// if the shape is not an amoeba,
// calculate the center point
// based on a rectangle,
// then rotate
// else
// use the xPt and yPt as
// the rotation point offset
// and then rotate
```

Not so fast: there's been a change! V

Changes for OOP approach

- Change `rotate` code just in class `Amoeba`
- (Hopefully) tested code for other classes remains **untouched**

Victory for OOP?

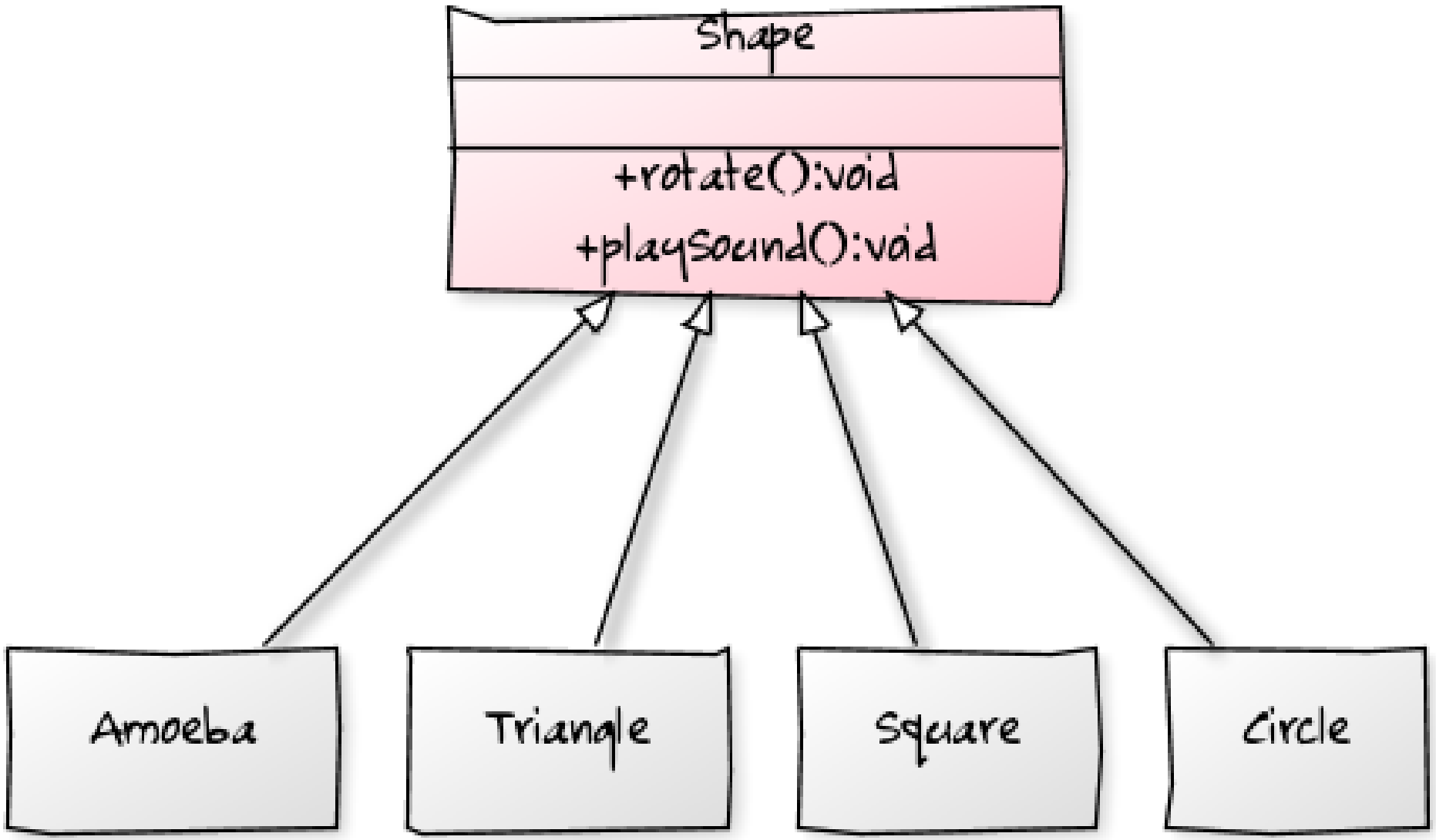
Victory for OOP?

- OOP solution much more maintainable
- But: duplicate code!
- Inheritance to the rescue!

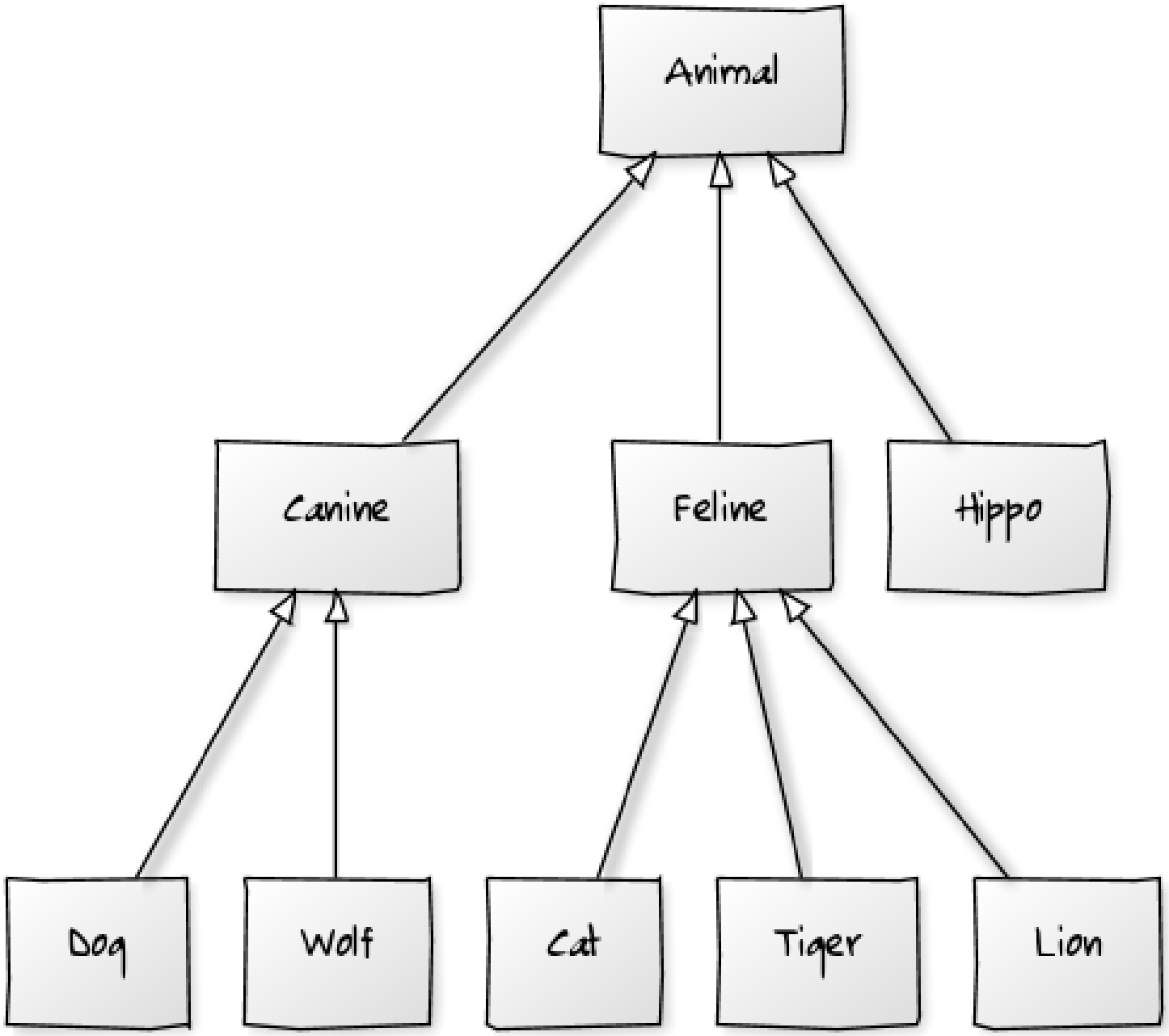
What do all four classes have in common?

- All classes represent shapes
- All shapes can be rotated and play a sound
- $\rightarrow$ let's move common features and put them into a new class: Shape

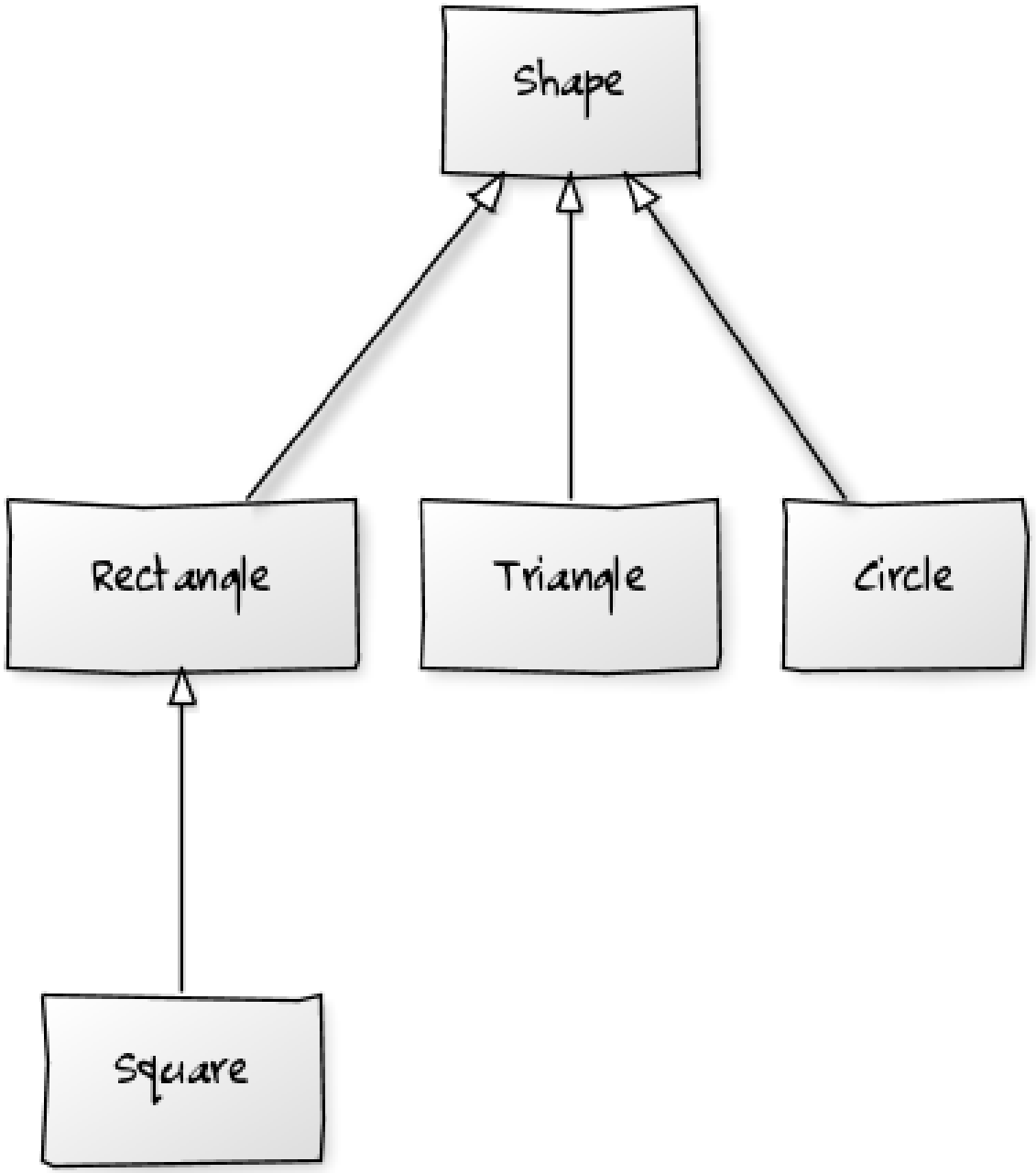
Graphical representation (UML)



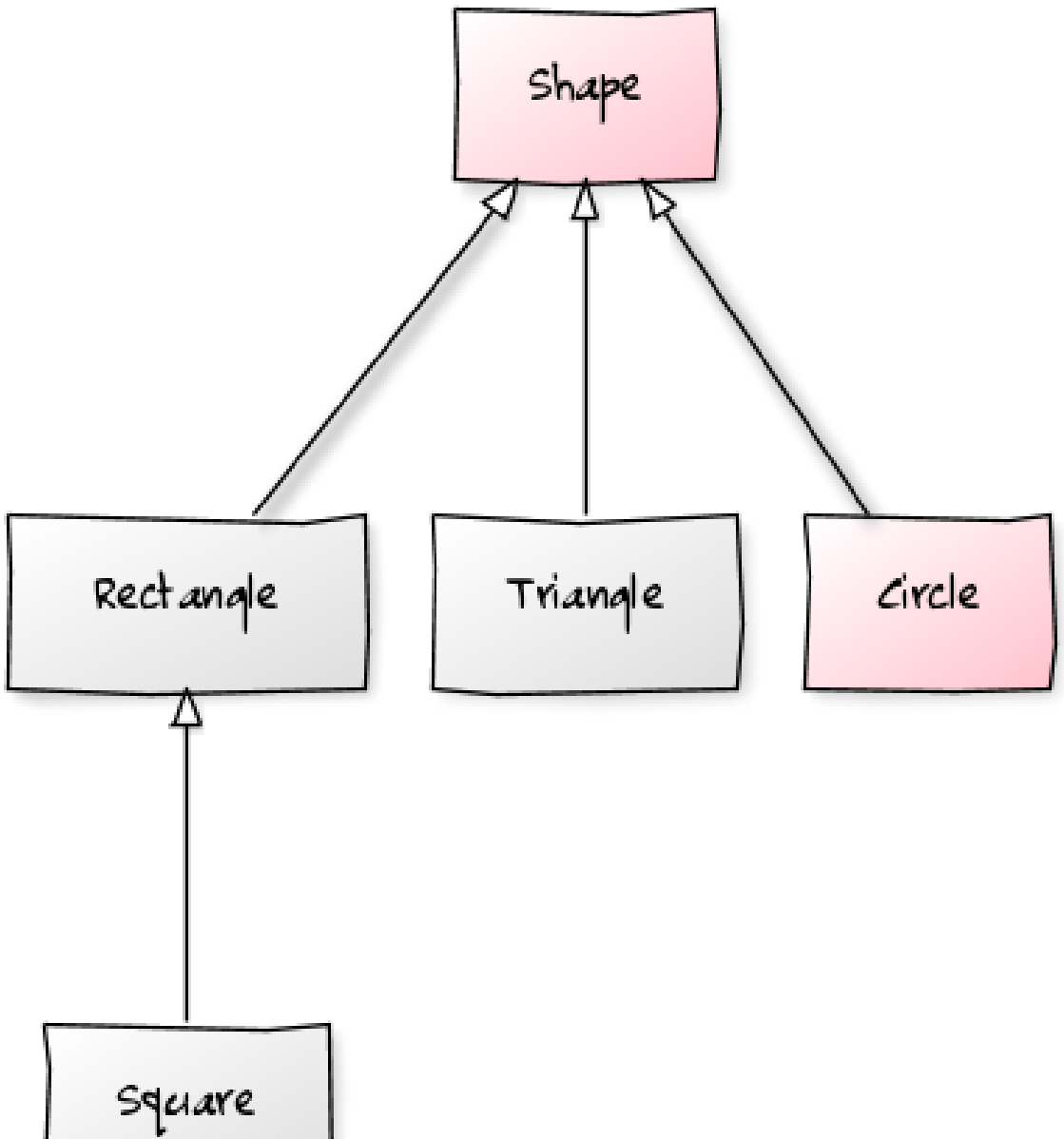
Class hierarchies – example: Animal



Class hierarchies – Example: Shape

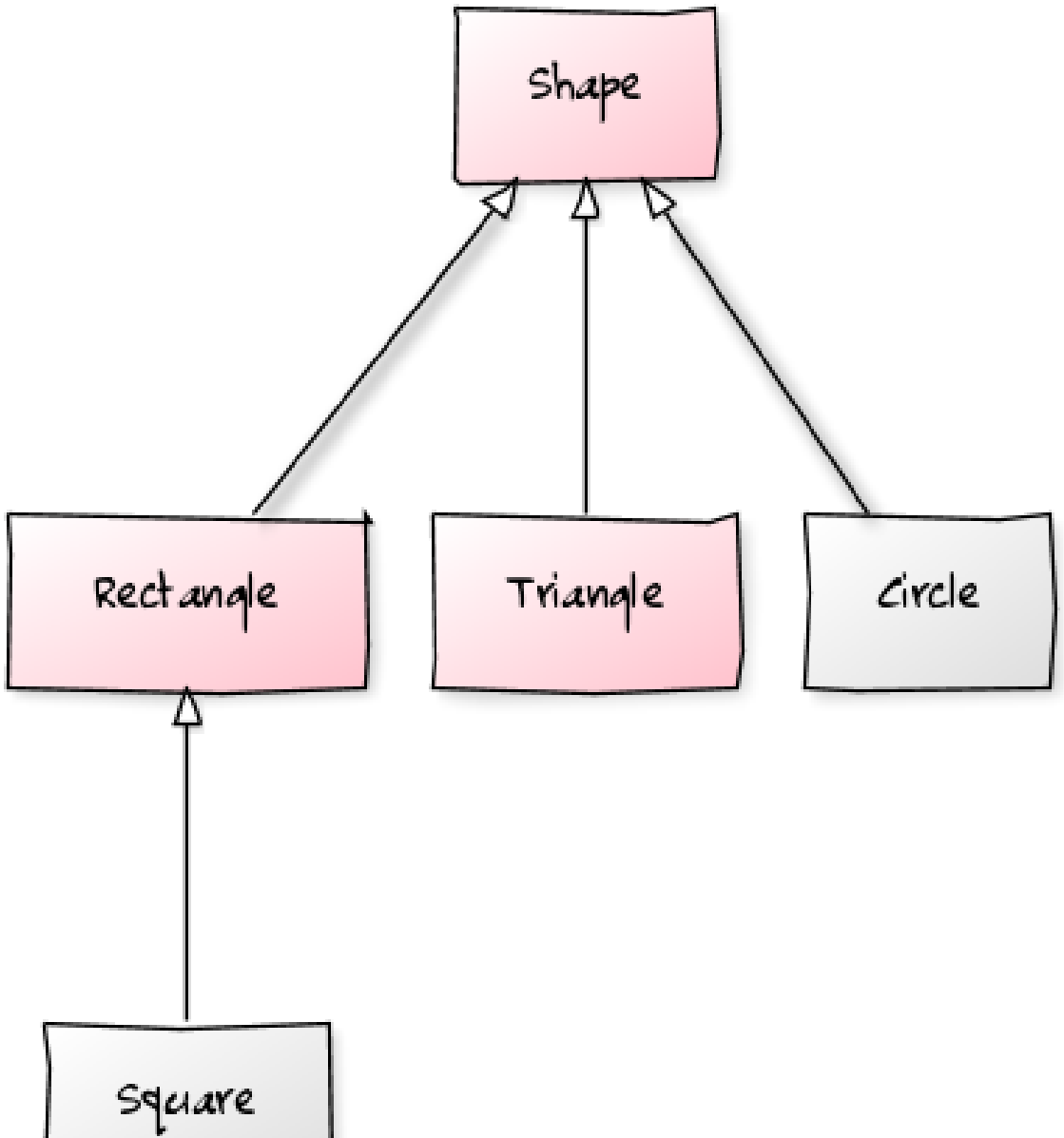


Class hierarchies – Example: Shape



Shape is the super class of Circle

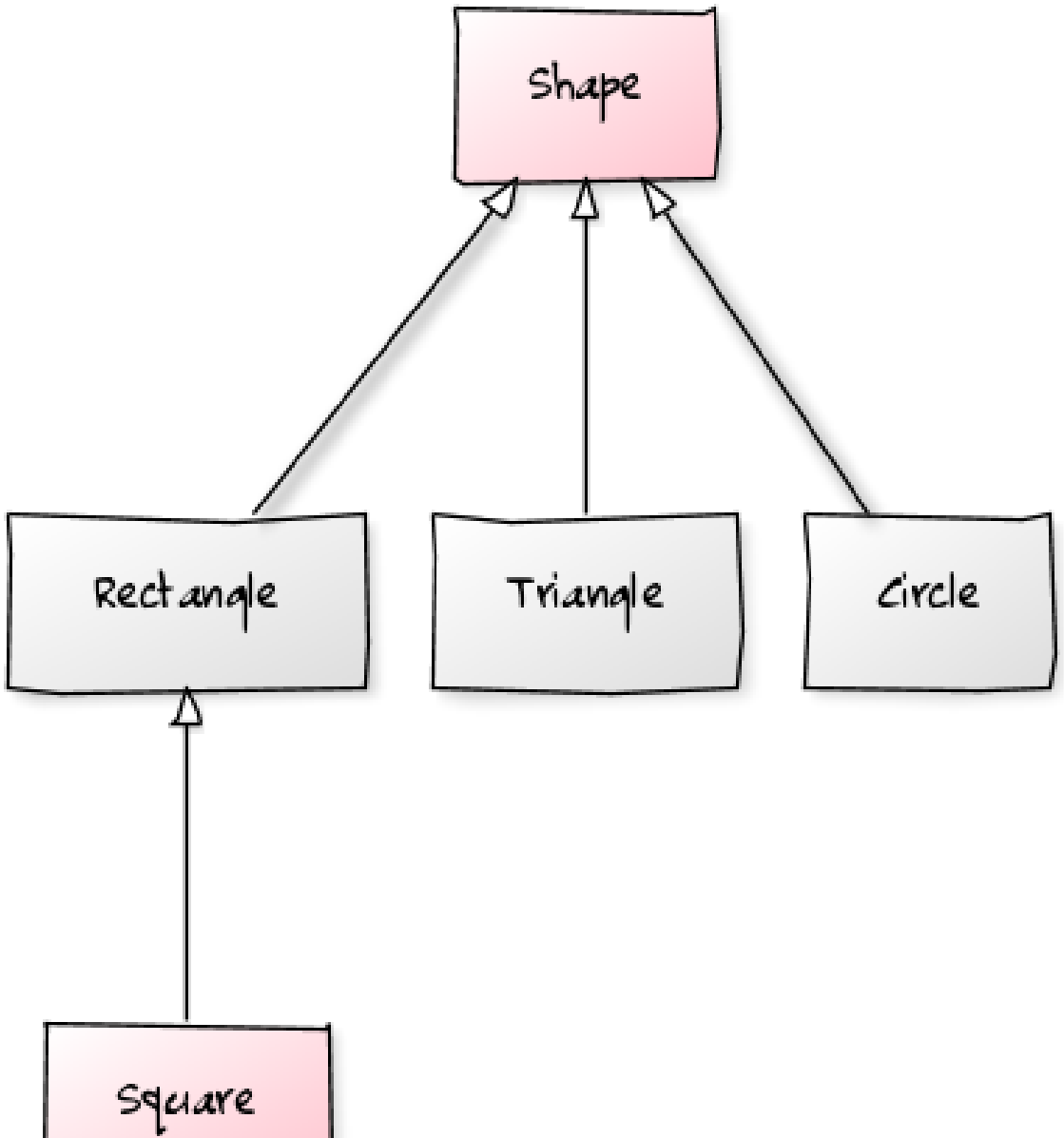
Class hierarchies – Example: Shape



Shape is the super class of Circle

and of Rectangle, and of a Triangle

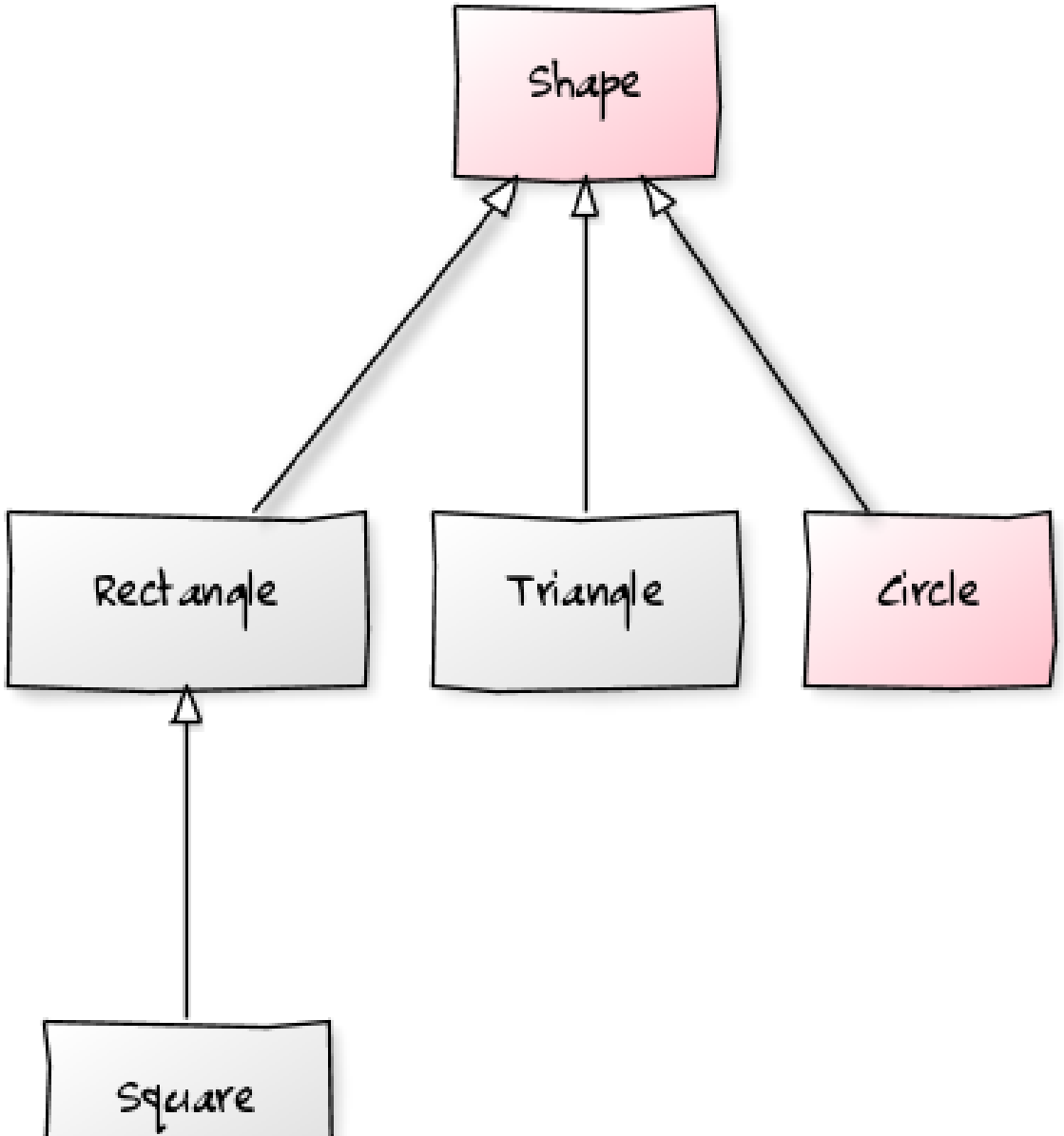
Class hierarchies – Example: Shape



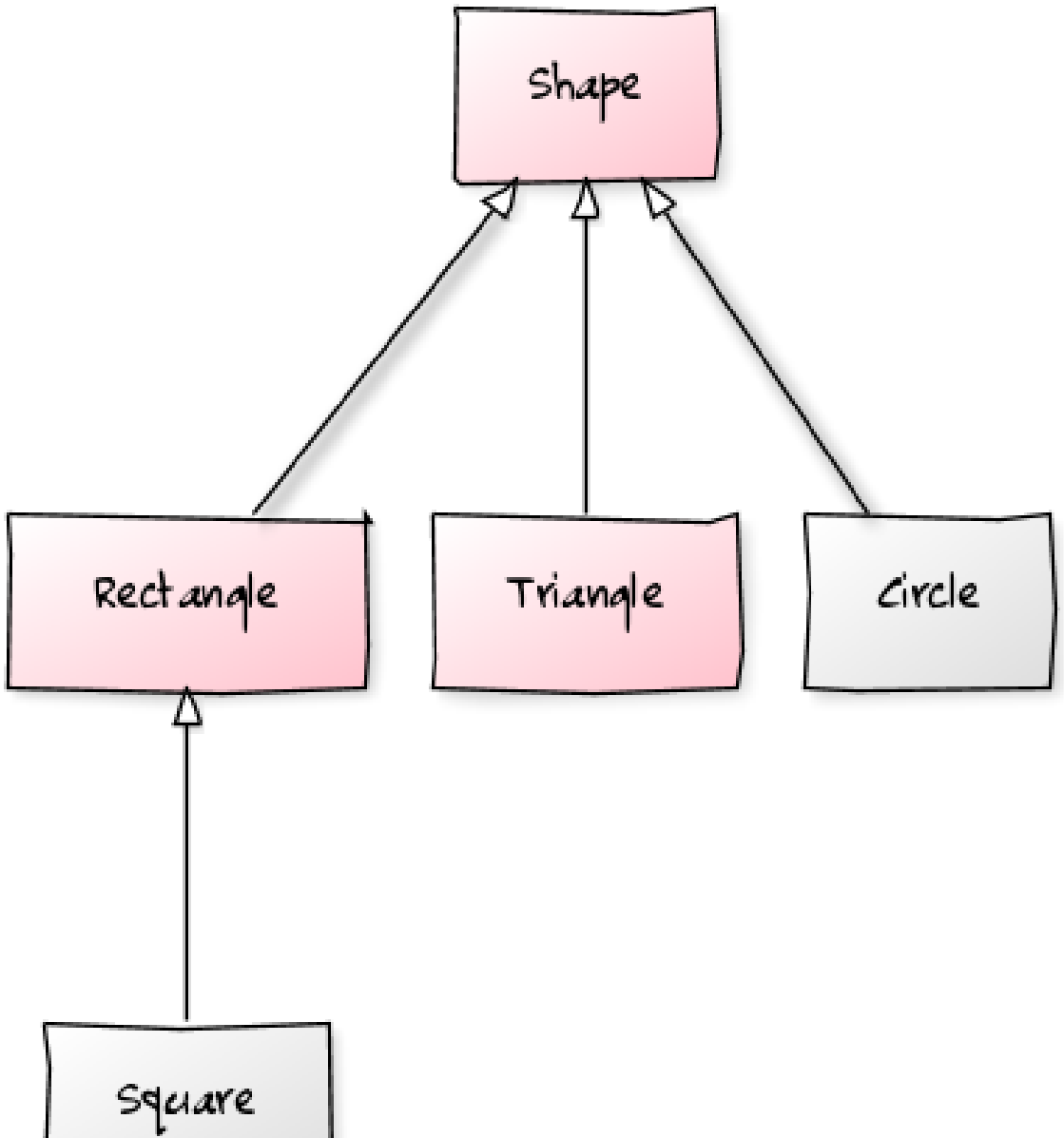
Shape is a super class of Square (transitive)

Class hierarchies – Example: Shape

Circle is a subclass of Shape



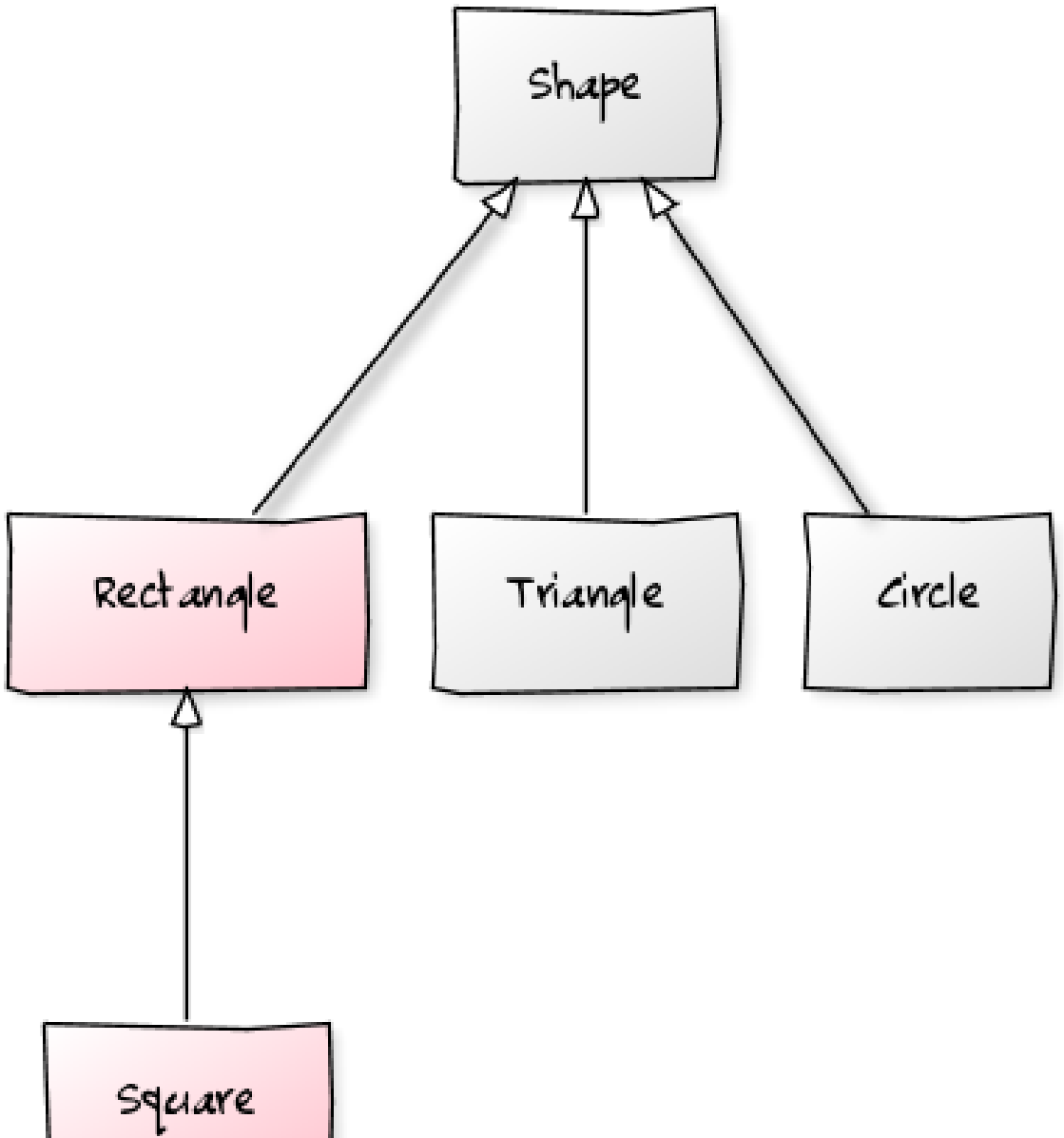
Class hierarchies – Example: Shape



Circle is a subclass of Shape

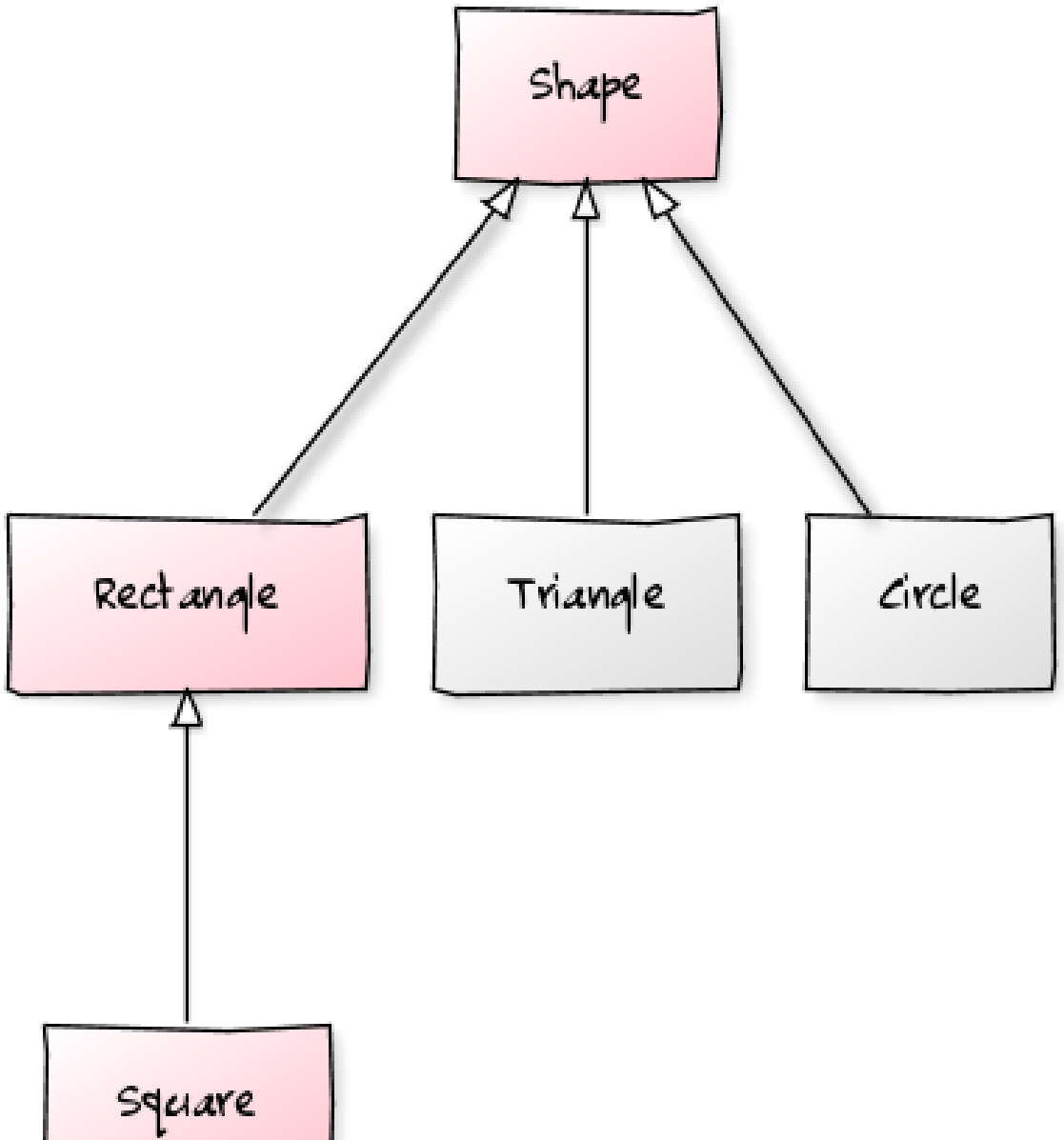
. . . such as Rectangle and Triangle

Class hierarchies – Example: Shape



Square is a direct subclass of Rectangle

Class hierarchies – Example: Shape



Square is a direct subclass of Rectangle
and a subclass of Shape

Class hierarchies

- Class hierarchies define an IS-A relation
- i.e.: <subclass> IS-A <superclass>, e.g.

Feline IS-A Animal	Circle IS-A Shape
Lion IS-A Feline	Rectangle IS-A Shape
Lion IS-A Animal	Square IS-A Rectangle

Class hierarchies

- Subclasses extend their superclass: they implement more specific or additional properties and behaviors
- Properties that are common to multiple classes are implemented in the super class
- **Advantage:** avoids duplicate code
  - less redundancy
  - changes at one place
  - effective for all subclasses (by inheritance...)

Class hierarchies

- Keyword `extends` in the "header" of a class definitions establishes an inheritance relation between two classes:

  ```
  class <subclass> extends <direct-superclass> { . . . }
  ```

Class hierarchies

- Keyword `extends` in the "header" of a class definitions establishes an inheritance relation between two classes:

  ```
  class <subclass> extends <direct-superclass> { . . . }
  ```

- Definition of the class `<subclass>`

Class hierarchies

- Keyword `extends` in the "header" of a class definitions establishes an inheritance relation between two classes:

  ```
  class <subclass> extends <direct-superclass> { . . . }
  ```

- Definition of the class `<subclass>`
- Keyword `extends`

Class hierarchies

- Keyword `extends` in the "header" of a class definitions establishes an inheritance relation between two classes:

  `class <subclass> extends <direct-superclass> { . . . }`

- Definition of the class `<subclass>`
- Keyword `extends`
- `<subclass>` is a subclass of `<direct-superclass>`:

  `<direct-superclass>`

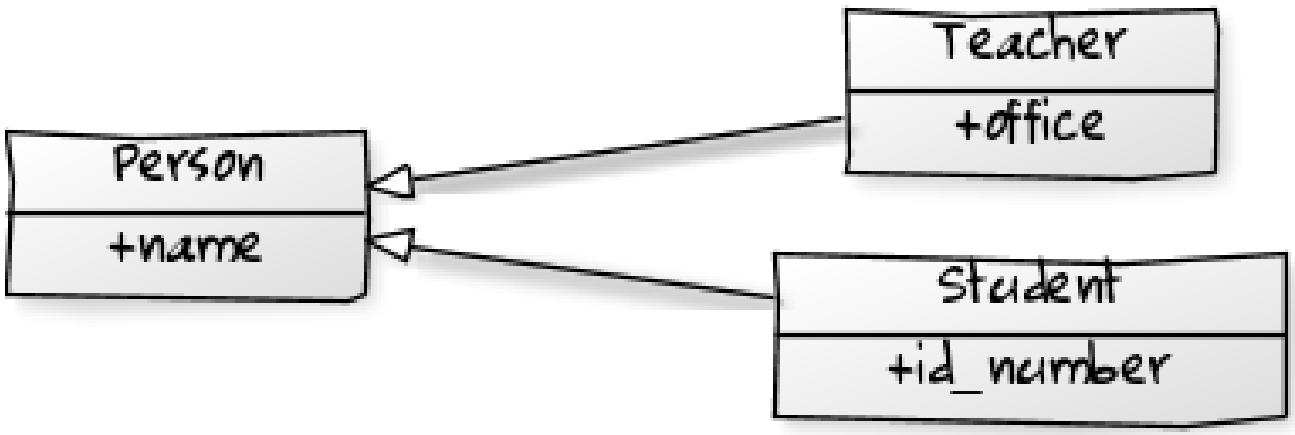
  `<subclass>`

Effects of Inheritance

- Subclass inherits functionality of the super class
  - Variables (fields)
  - Methods
- only visible elements are inherited!

Example: Person – Student – Teacher

Class diagram



- Person has a name
- Student has an ID number
- Teacher has an office

Inheritance

```java
public class Person {
  private String name;

  public Person( String name ) {
    this.name = name;
  }

  public String getName() {
    return this.name;
  }
}
```

Person.java

Inheritance

```java
public class Student extends Person {
  private int idNumber;

  public Student( String name, int idNumber ) {
    super( name ); // call to Person constructor
    this.idNumber = idNumber;
  }

  public int getIdNumber() {
    return this.idNumber;
  }
}
```

Student.java

Inheritance

```java
public class Teacher extends Person {
  private int office;

  public Teacher( String name, int office ) {
    super( name );
    this.office = office;
  }

  public int getOffice() {
    return this.office;
  }
}
```

Teacher.java

Inheritance

```
public class PersonTest {

    public static void main( String[] args ) {

        Student stud = new Student( "Fritz", 12345 );
        System.out.println(stud.getName());//Pers.method!

        Teacher thb = new Teacher( "Thomas", 122 );
        System.out.println(thb.getName() + ". Office: " + thb.
            getOffice() );
    }
}
```

PersonTest.java

Inheritance

```
public class FaultyTeacher extends Person {
    private int office;

    public FaultyTeacher( String name, int office ) {
        super( name );
        // does not compile (name is private in Person)
        this.name = "Prof. " + this.name;
        this.office = office;
    }

}
```

FaultyTeacher.java – does not compile!!

Inheritance

```
public class FaultyTeacherPatched extends Person {
    private int office;
    private String name; // hides Person's name

    public FaultyTeacherPatched(String name, int office) {
        super( name );
        this.name = "Prof. " + name;
        this.office = office;
    }

    public static void main( String[] args ) {
        FaultyTeacherPatched t = new FaultyTeacherPatched( "XY"
            , 505 );
        System.out.println( t.getName() ); // prints Person's
            name
        System.out.println( t.name ); // prints FTP's name
    }
}
```

FaultyTeacherPatched.java – evil!

Inheritance

- `super()` calls the constructor of the super-class
- Visible elements are inherited (`getName()`)
- `private` elements are invisible, even for sub-classes (`name`)
- "Hiding": variables can be hidden (`name` in `FaultyTeacherPatched`)
- "Overriding": methods can be overwritten ...

Inheritance

```
public class Oftp extends Person {
  private int office;
  private String name; // hides Person's name

  public Oftp( String name, int office ) {
    super( name );
    this.name = "Prof. " + name;
    this.office = office;
  }

  public String getName() { // ovrrides meth in Pers.
    return this.name;
  }

  public static void main( String[] args ) {
    Oftp t = new Oftp( "XY", 505 );
    System.out.println( t.getName() );
  }
}
```

OverridingFaultyTeacherPatched (Oftp.java) – still bad style

Inheritance

```
public class OverridingTeacher extends Person {
  private int office;

  public OverridingTeacher(String name, int office) {
    super( name );
    this.office = office;
  }

  public String getName() {
    return "Prof. " + super.getName();
  }

  public static void main( String[] args ) {
    OverridingTeacher t = new OverridingTeacher("XY",505);
    System.out.println(t.getName());
  }
}
```

OverridingTeacher.java

@Override annotation

- Write @Override before each method you are overriding
- Compiler error if you fail to override (this often happens when you make a typo - you would get the parent method then!)
- Good style (but code will compile without it)

Inheritance

```java
public class OverridingTeacher extends Person {
  private int office;

  public OverridingTeacher(String name, int office) {
    super( name );
    this.office = office;
  }

  @Override
  public String getName() {
    return "Prof. " + super.getName();
  }

  public static void main( String[] args ) {
    OverridingTeacher t = new OverridingTeacher("XY",505);
    System.out.println(t.getName());
  }
}
```

OverridingTeacher.java

Inheritance

- `super. ...` allows access to (visible) elements of the super-class
- "Overriding": methods can be overwritten in a sub-class
- Overriding is not Hiding:
  - Hidden fields can be made visible again by type casting
  - Overwritten methods remain associated with the object of the sub-class
- Hiding is rarely (never?) necessary – instead: new variable and/or overriding

Outline

Packages

- Packages group classes belonging together thematically, e.g.
- `java.io` for system input and output
- `java.util` for (very!) useful methods and tricks (Collections, Date & Time, ...)
- `de.uniheidelberg.cl.project` for custom packages in "project"

Packages

- Naming schema: `de.uniheidelberg.cl.project`
- Directory structure represents hierarchical structure:
  `xyz/de/uniheidelberg/cl/project/Mention.java`
- in order to use a package, the (top-level) directory containing the package (here: `xyz`) needs to be specified with the java(c) option -classpath (-cp):

```
> javac -cp .:xyz xyz/de/uniheidelberg/cl/project/Mention.java
> java -cp .:xyz de.uniheidelberg.cl.project.Mention
```

Packages

- Naming schema: de.uniheidelberg.cl.project
- Directory structure represents hierarchical structure:
  xyz/de/uniheidelberg/cl/project/Mention.java
- in order to use a package, the (top-level) directory containing the package (here: xyz) needs to be specified with the java(c) option -classpath (-cp):

```
> javac -cp .:xyz xyz/de/uniheidelberg/cl/project/Mention.java
> java -cp .:xyz de.uniheidelberg.cl.project.Mention
```

Packages

- Naming schema: de.uniheidelberg.cl.project
- Directory structure represents hierarchical structure:
  xyz/de/uniheidelberg/cl/project/Mention.java
- in order to use a package, the (top-level) directory containing the package (here: xyz) needs to be specified with the java(c) option -classpath (-cp):

```
> javac -cp .:xyz xyz/de/uniheidelberg/cl/project/Mention.java
> java -cp .:xyz de.uniheidelberg.cl.project.Mention
```

Packages

- Naming schema: `de.uniheidelberg.cl.project`
- Directory structure represents hierarchical structure:
  `xyz/de/uniheidelberg/cl/project/Mention.java`
- in order to use a package, the (top-level) directory containing the package (here: `xyz`) needs to be specified with the java(c) option -classpath (-cp):

```
> javac -cp .:xyz xyz/de/uniheidelberg/cl/project/Mention.java
> java -cp .:xyz de.uniheidelberg.cl.project.Mention
```

Packages

- Naming schema: `de.uniheidelberg.cl.project`
- Directory structure represents hierarchical structure:
  `xyz/de/uniheidelberg/cl/project/Mention.java`
- in order to use a package, the (top-level) directory containing the package (here: `xyz`) needs to be specified with the java(c) option -classpath (-cp):

```
> javac -cp .:xyz xyz/de/uniheidelberg/cl/project/Mention.java
> java -cp .:xyz de.uniheidelberg.cl.project.Mention
```

Packages

- Assignment of a class to a package needs to be stated explicitly
- `package de.uniheidelberg.cl.project;`
- package statement needs to be the first line in the source coed of the class
- Only one package statement per class

Packages

Using a class from another package

- Fully specified location: `de.uniheidelberg.cl.project.Mention x = new ...`
- Import the package

package-private access

- Previously: `private, public`
- Elements can also be package-private
- without visibility modifier = package-private
- package-private means
  - within the same package: like `public`
  - outside of the package: like `private`
  - → visible to all classes in the same packages, invisible to all other classes

Outline

protected access

- previously: `private`, `public`
- `protected`
  - `protected` elements are visible to sub-classes only
  - for all other classes: like `private` (invisible)
  - allows direct access to details of the implementation for sub-classes

protected access and final

- "malicious" sub-class could exploit `protected` access (e.g. change invariants)
- this can be avoided by declaring classes as `final`
  - `final` classes prohibit sub-classing:
    `... extends <final-class>` is not allowed
  - `final` methods cannot be overridden
  - `final` variables cannot be changed

General advice

`private`, `protected` or `public`

- Best approach: leave fields `private`
- If underlying implementation changes: sub-classes might break
- Add controlled access to inheritors by using `protected` methods

What you should know after this session

- How to draw class diagrams
- Motivation for Inheritance
- Implications of Inheritance
- Hiding & overriding
- Designing class hierarchies
- Use of packages and `package-private`
- `protected`, `final`

Literature


The Java Tutorials
`http://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html` *and* `override.html`.


Sierra, K. & Bates, B.
Head First Java. (Mostly Ch. 2)
O'Reilly Media, 2005.


Ullenboom, Ch.
Java ist auch eine Insel. (Ch. 5)
Galileo Computing, 2012.


Eckel, B. (for reference)
Thinking in Java. (Ch. 6)
Prentice Hall, 2006.