

Languages and Tools for Object-Oriented Development & Object-Oriented Analysis and Design

Rainer Marrone based on slides of
Prof. Dr. J.W. Schmidt
Dr. Hans-Werner Sehring

Software Systems Institute (STS)

Harburger Schloßstraße 20
21071 Hamburg

<http://www.sts.tu-harburg.de/teaching/>

Software Development Today

- in teams
- by tool support
- under reuse of previous investments
- with change management in mind

Course Concept

Course Contributions	Analysis	Design	Implementation	LTOOD	OOAD	LabClass
<i>Basic Concepts</i>	Interviews: - Goals - Resources - Priorities, ...	Graphs, Nets: - ER - Activities - Sequences, ...	Structured Code: - Classes - Modules - Relations, ...	*	*	
<i>Languages (Syntax, Semantics)</i>	Natural Language Texts, (Diagrams)	(Semi-)Formal Language Diagrams (UML)	Formal Language PL Code (<i>Java</i> ®, SQL, ...)	*	(*)	*
<i>Tools (overview, use, interop., ...)</i>	<i>Word</i> ®, Pen & Paper, Knowledge Mgmt., (<i>Together</i> ®)	<i>Together</i> ®, (<i>Word</i> ®), Libraries, ...	<i>Eclipse</i> ®, various development tools	*		*
Pragmatics ("how to ...")	Cases, Lessons learnt, Principles, Risks, ...				*	

Modern software development and its integration with the course.

Course Outline, Timetable

Date	Topic	OOAD	LTOOD
24 Oct	Object-Orientation I	✓	
31 Oct	Object-Orientation II	✓	
7 Nov	Development Process & Analysis Phase	✓	
14 Nov	Design Phase I, Overview	✓	
21 Nov	Languages and Tools Overview		✓
28 Nov	Modeling Tools (e.g., for UML)		✓
5 Dec	Design Phase II, Details	✓	
12 Dec	Implementation Phase		✓
19 Dec	OO Libraries and Extensions		✓
9 Jan	Version & Change Tracking Tools		✓
16 Jan	Integrated Development Environments		✓
23 Jan	<i>"Industrial Position"</i>	✓	✓
30 Jan	Testing and Documentation Tools		✓
6 Feb	Software Development Projects, Summary	✓	

Exercises and Lab Classes

- Joint for OOAD and LTOOD
- Two shifts, several groups of 4 persons each
- First exercise on 31 Oct 2005
- Location: HS20, room 214
 - Also see: <http://www.sts.tu-harburg.de/institute/sts-map.html>
- Please register for the exercise groups.

Contact persons:

Miguel Garcia

miguel.garcia@tu-harburg.de

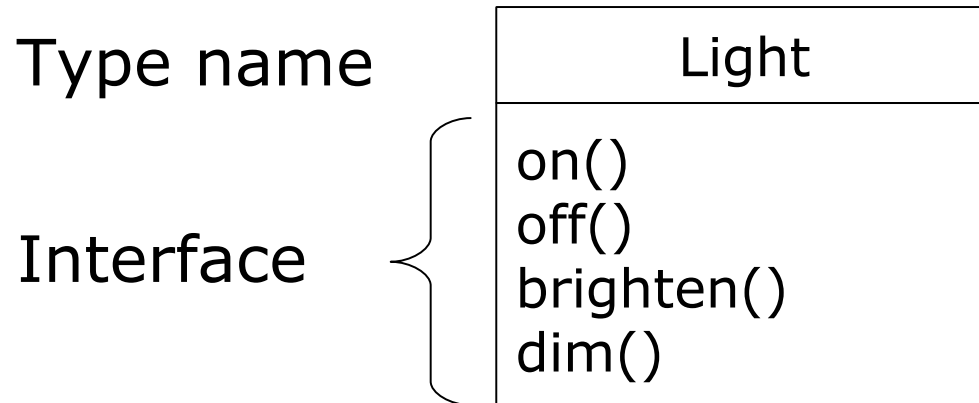
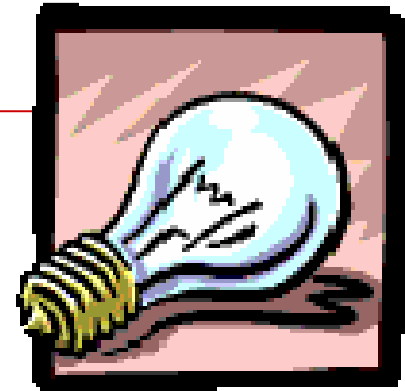
Exam

- Individual exams for LTOOD and OOAD
- At the end of the term
- In written form
- Closed-book

Object Orientation I

Introduction

What is an Object?



Objects couple characteristics and relationships with behaviour

Object creation: `Light lt = new Light ();`

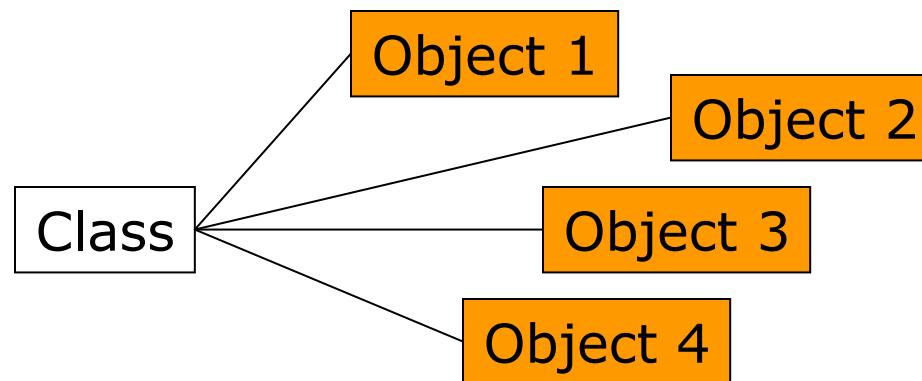
Message passing: `lt.on ();`

Alan Kay's 5 Rules for OO

- Everything is an object.
- Each object has its own memory made up of other objects.
- Every object has a type.
- All objects of a particular type can receive the same messages.
- A program is a bunch of objects telling each other what to do by sending messages.

Classes

- A class defines the methods and attributes (together called members) shared by all objects of a certain kind.
- In other words: A class sums up the commonalities of all objects belonging to the class.



many objects can belong to the same class

Class – Instance – Object

- When you create an object:
`Light lt = new Light ();`
you create an instance of the class called `Light` and bind it to the variable `lt`.
- An instance is a specific occurrence of a member of its class.
- Object is a synonym for instance.

Example: Your car is one of many cars. It is an instance of the class `Car`.

Example of a Simple Class

```
class Rectangle {  
    int x; int y;  
    int width; int height;  
    void draw() {  
        // ...  
    }  
    void erase() {  
        // ...  
    }  
}
```

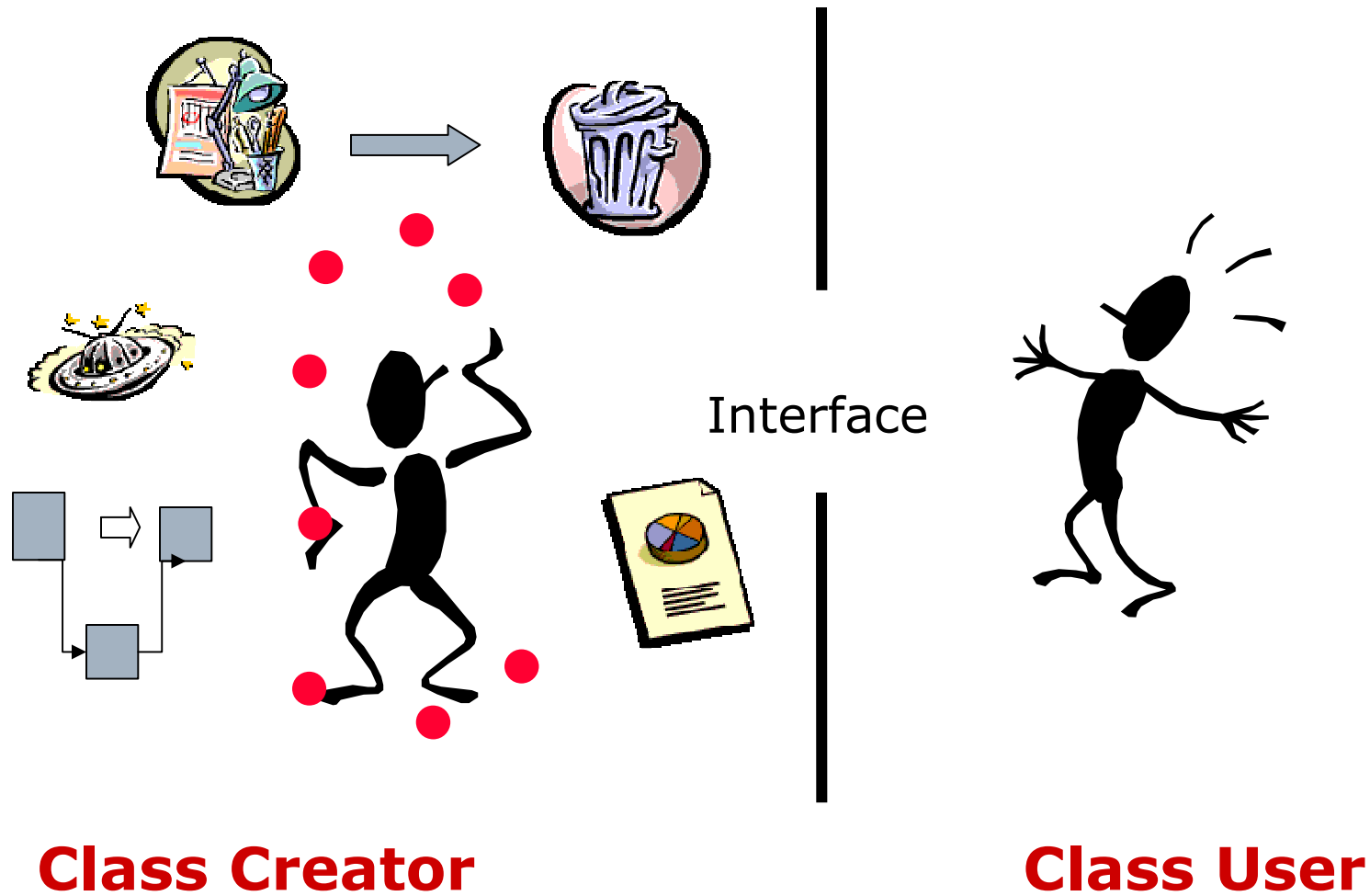
box : Rectangle

x = 100; y = 200;
width = 10;
height = 20;

A class
instance

A class
definition

Objects Hide Complexity



Example of Hidden Complexity

Creator of the class ...

```
class Person {  
    double height ;  
    double weight ;  
    double calculateBMI () {  
        return weight / (height * height) ;  
    }  
}
```

... needs to know the details about the body-mass-index

User of the class has a very easy life.

```
Person p = new Person() ;  
p.height = 1.80 ; p.weight = 75 ;  
p.calculateBMI () ;
```

Object Orientation

Approach and Concepts

The Object-oriented Approach

“The OO approach has been exploited by a wide range of applications (user interface construction, operating systems, databases, ...).

Why is it so successful?

1. Close *Relationship* between software models and physical systems
2. Improved *Resilience* of the software models
3. Increased *Reusability* of the components of the software models (“*the 3 Re’s*”).

[Abadi, Cardelli: A Theory of Objects]

1. Relationship between SWM and PM

The **analogy** with a physical model (PM) can be useful in the development of a software model (SWM)
(→ object-oriented *analysis* process).



(see: Simula [Dahl&Nygaard] as the root of OO)

2. Resilience of SWM

Resilience of design in the face of changes is a consequence of building abstractions. Objects form natural data abstraction boundaries and help focus a design on system structure instead of algorithms.

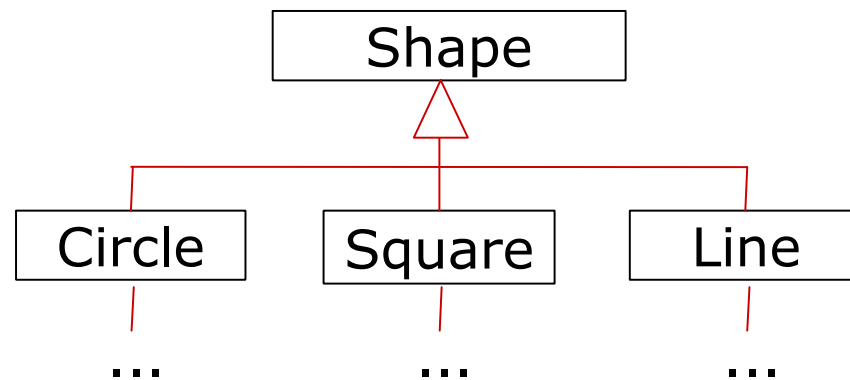
Algorithms are factored into methods that are attached to objects, making the objects **behaviorally autonomous**.

Because of object abstractions, a design can evolve with fewer pervasive reorganizations (→ object-oriented *design* process).

Light
on() off() brighten() dim()

3. Reusability of SWMs

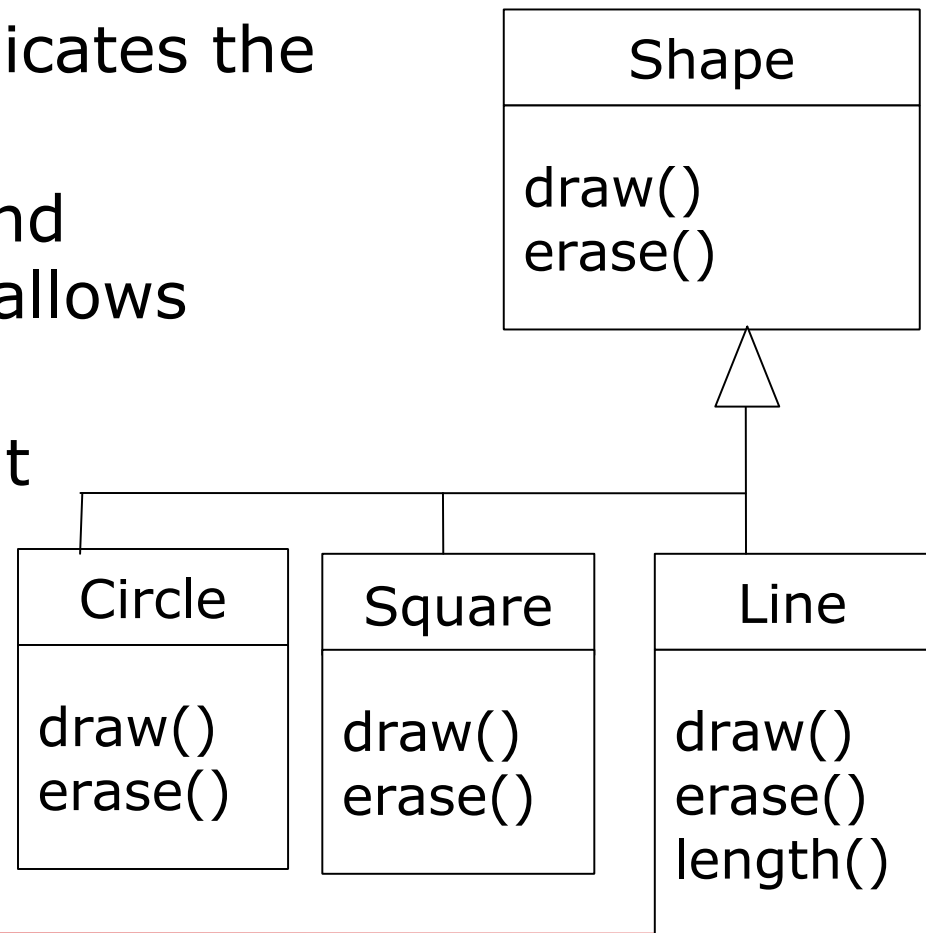
Objects are naturally organized in **taxonomies** during analysis, design, and implementation. This hierarchical organization encourages the reuse of methods and data that are located higher in the hierarchy (→ object-oriented *implementation* process).“



Concept: Inheritance I

Inheritance ...

- ❑ ... automatically duplicates the interface
- ❑ ... reuses interface and implementation but allows extension
- ❑ ... relates classes that share a part of their definition (therefore each Circle, e.g., is still a Shape).



Inheritance II

Inheritance allows:

- redefinition of inherited methods (`draw()`)
 - same method name
 - same argument type list
 - same return type
- addition of new methods (`length()`)
- addition of new data members



method
overriding

Concept: Overriding

- ❑ You might need to do things differently in a sub-class.
- ❑ Supply a new implementation of a method.

Deduce Meaning from Context

- You say:
 - Wash the car.
 - Wash the shirt.
 - Wash the dog.
- You don't say:
 - carWash the car.
 - shirtWash the shirt.
 - dogWash the dog.



Concept: Method Overloading

- ... computers can also figure out the context:

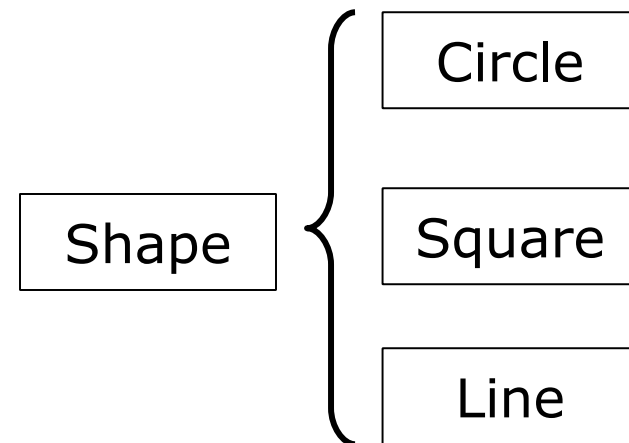
```
class SuperWasher {  
    void wash (Shirt s) { ... }  
    void wash (Car c) { ... }  
    void wash (Dog d) { ... }  
}
```

- This is called overloading
 - substantially different from overriding!
 - **overloading** is about "name economy"
 - **overriding** is about "object autonomy"
- Not special to OO, present in other paradigms as well

Goal: Safe and Flexible Substitutability

- One piece of code defined for objects of a (super-) class also works for all objects of its sub-classes:
- It does not matter whether `s` is actually a Circle, Square or Line.

```
Shape s ;  
// ...  
s.draw () ;  
s.erase () ;
```

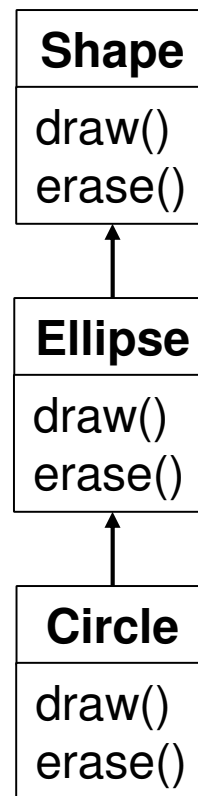


Means: Polymorphism

- An object can appear as any of all its ancestors

An Ellipse is a Shape.

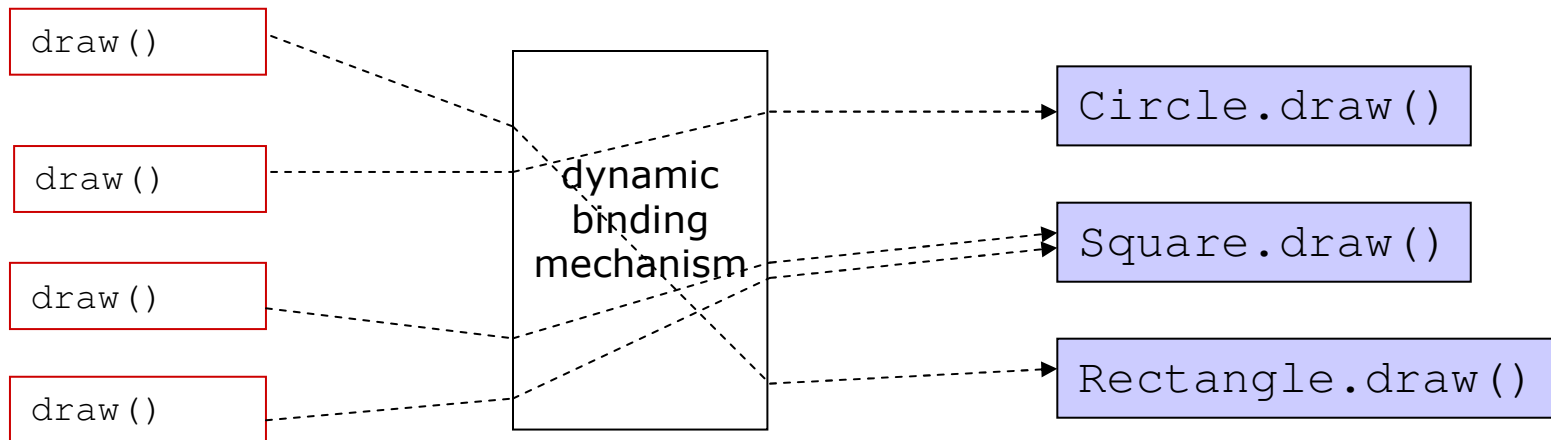
A Circle is an Ellipse.
A Circle is also a Shape.



[*Monomorphism*: An Object is related to exactly one type (class).]

Means: Dynamic Binding

- Which class definition to be related (i.e., which implementation to be called) is determined at runtime based on the class used for object creation.



- This is essential for safe and flexible substitutability.

Example: Writing General Code

```
void doStuff (Shape s) {  
    s.erase () ;  
    // ...  
    s.draw () ;  
}  
// ...  
Circle c = new Circle () ;  
Triangle t = new Triangle () ;  
Line l = new Line () ;  
doStuff (c) ;  
doStuff (t) ;  
doStuff (l) ;
```

Each of the objects c, t and l appears to the doStuff method as a Shape.

This way, you have to write the code for doStuff only once even though you have many sub-classes of Shape.

Variability and Polymorphism (1)

- Substitutability requires careful consideration of the variation of members
- Return values of methods may be more special in subclasses (covariance):
 - `Circle createShape ()` may replace
 - `Shape createShape ()`
 - **Usage:**
`Shape s=createShape () ; //expects "Shape"`

Variability and Polymorphism (2)

- Parameters of methods have to be more general in subclasses (contravariance):
 - `rotate(Shape s)` may replace
 - `rotate(Circle c)`
 - Usage:
`rotate (new Circle ())`
- Types of variables can not be changed (invariance):
 - `class Shape { public Color color ; ... }`
 - color as l-value:
`new Shape ().color = Color.black ;`
 - color as r-value:
`Color defaultColor = new Shape ().color ;`

Reuse

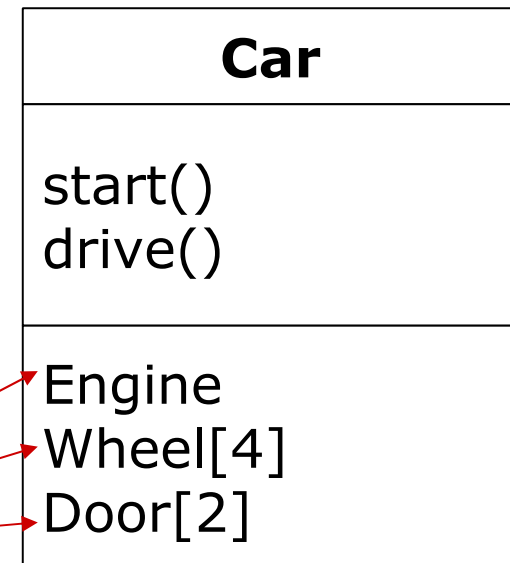
When you need a class, you can:

1. Get the perfect one off-the-shelf (one extreme).
2. Write it completely from scratch (the other extreme).
3. Reuse an existing class with composition.
4. Reuse an existing class with inheritance.

Reuse by Composition



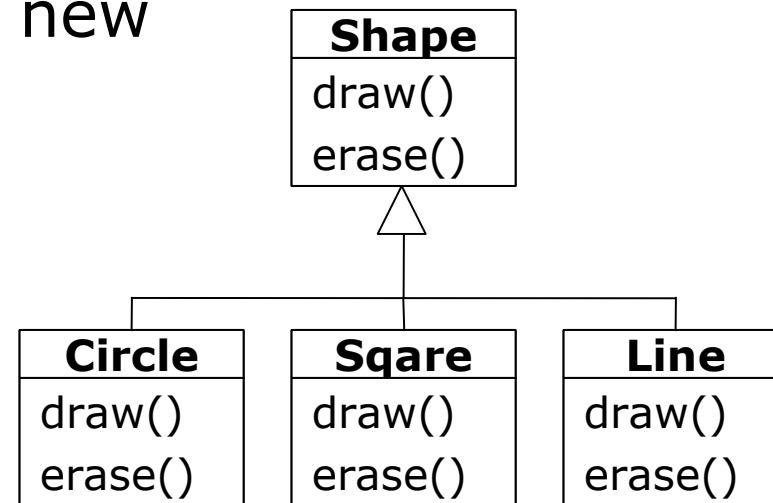
- Make new classes by composing existing ones
- Composition by "Has-A" or "Part-Of" relationships
- Instances of existing classes as data members



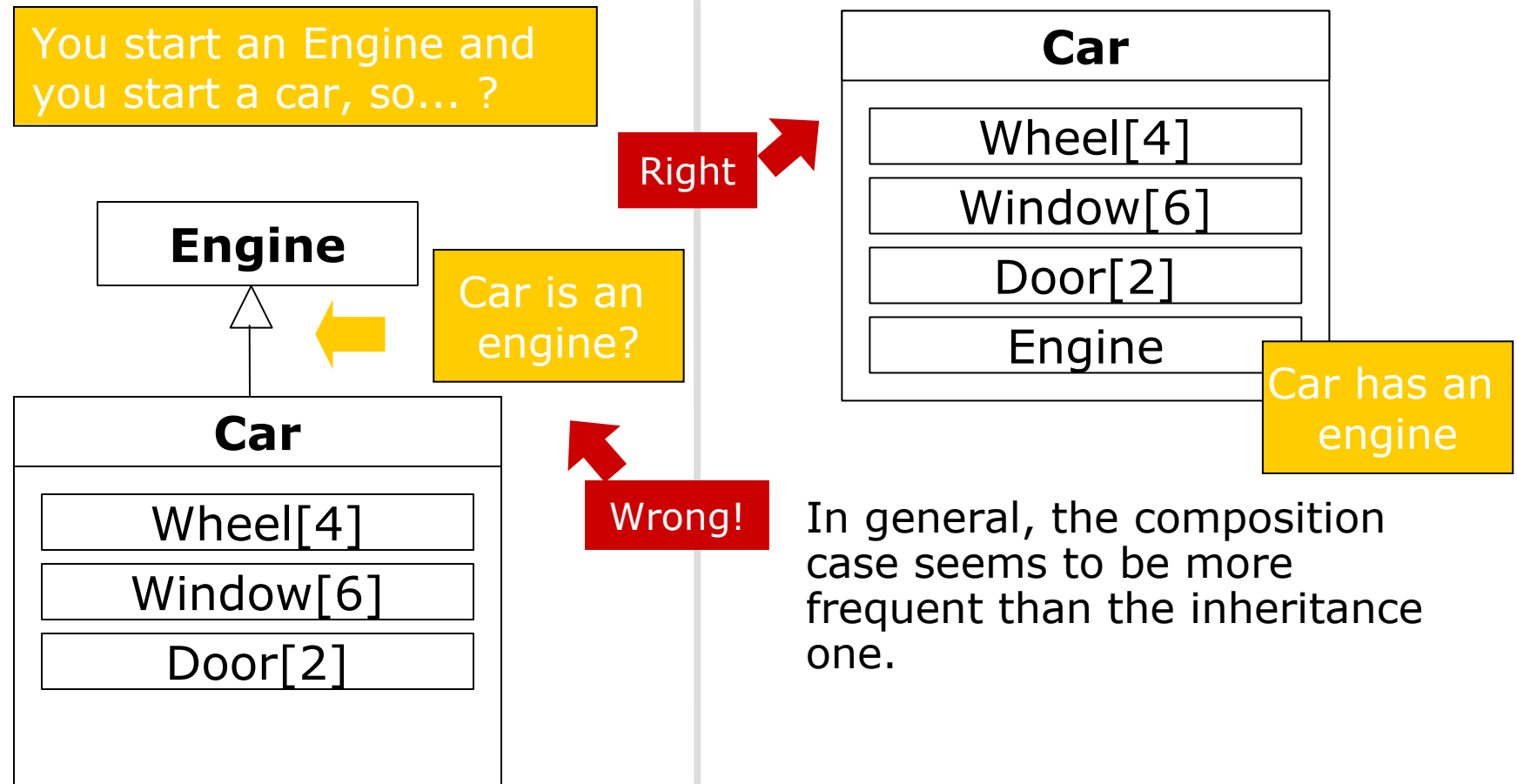
Car has engine, wheels and doors. But these are implemented elsewhere.

Reuse by Inheritance

- Make new classes by extending existing ones
- Inheritance by “Is-A” relationship
- You only add what is new or changed in the subclasses.



Inheritance vs. Composition?

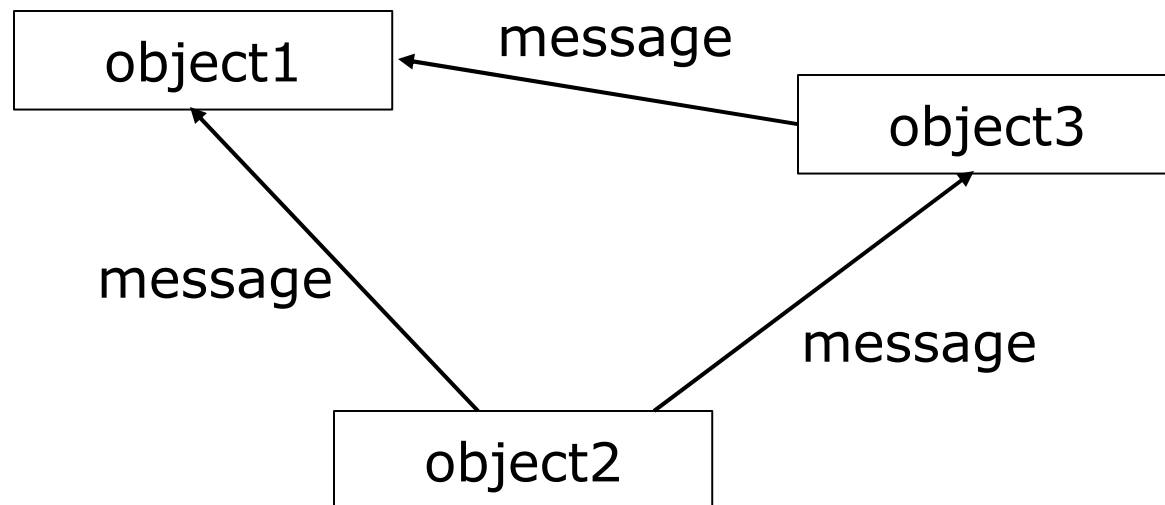


Object Orientation

Working with Objects

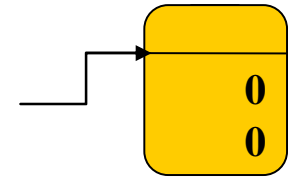
Object-Oriented Programs

- ❑ OO-Programs are essentially made up of objects
- ❑ These objects send messages to each other.
- ❑ On message receive objects execute methods.

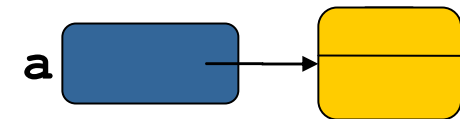


References

- A reference identifies an object. However, objects by themselves are anonymous.



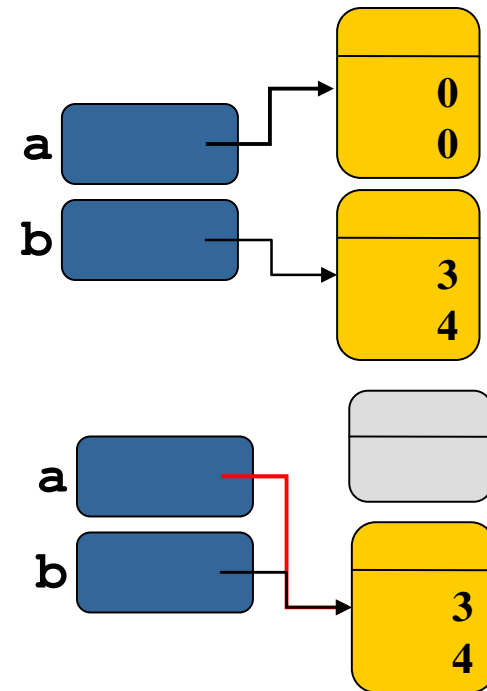
- The reference can be bound to a variable (assignment).



- Note that this does not mean that the variable is the reference or is the object!

Reference Type Variables

- Two **variables** might reference different **objects**...
- ... but may also reference same object
- When you say $a := b$, both reference the same instance.
- If you change the object referenced by a , you also change that referenced by b !



Always make sure, that you know which instance you are talking about!

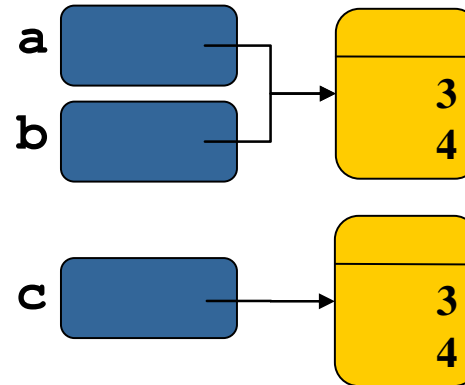
Comparison

- Comparison by **reference** (identity)

- `a==b`
- `c!=a`

- Comparison by **value**

- `a.equals(c)`
- `a.equals(b)`

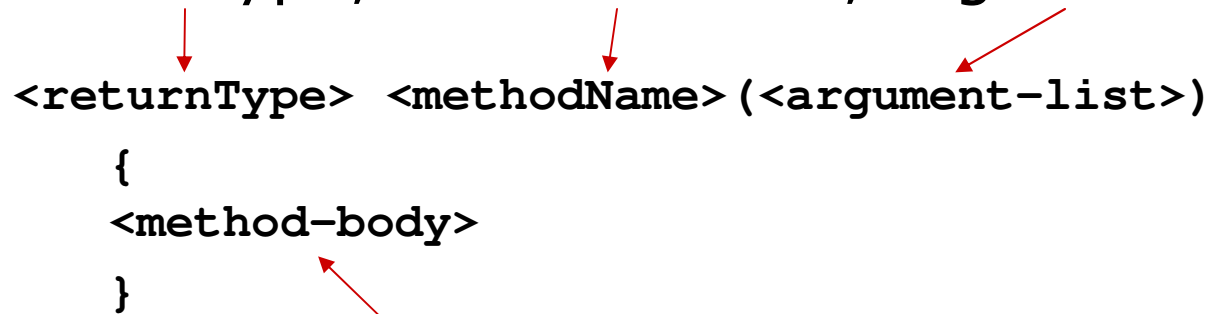


- If you create your own class, the behavior of `equals()` is to compare references

Methods

- Operations on objects
- Defined inside classes
- Composed of method signature
(return type, method name, arguments)...

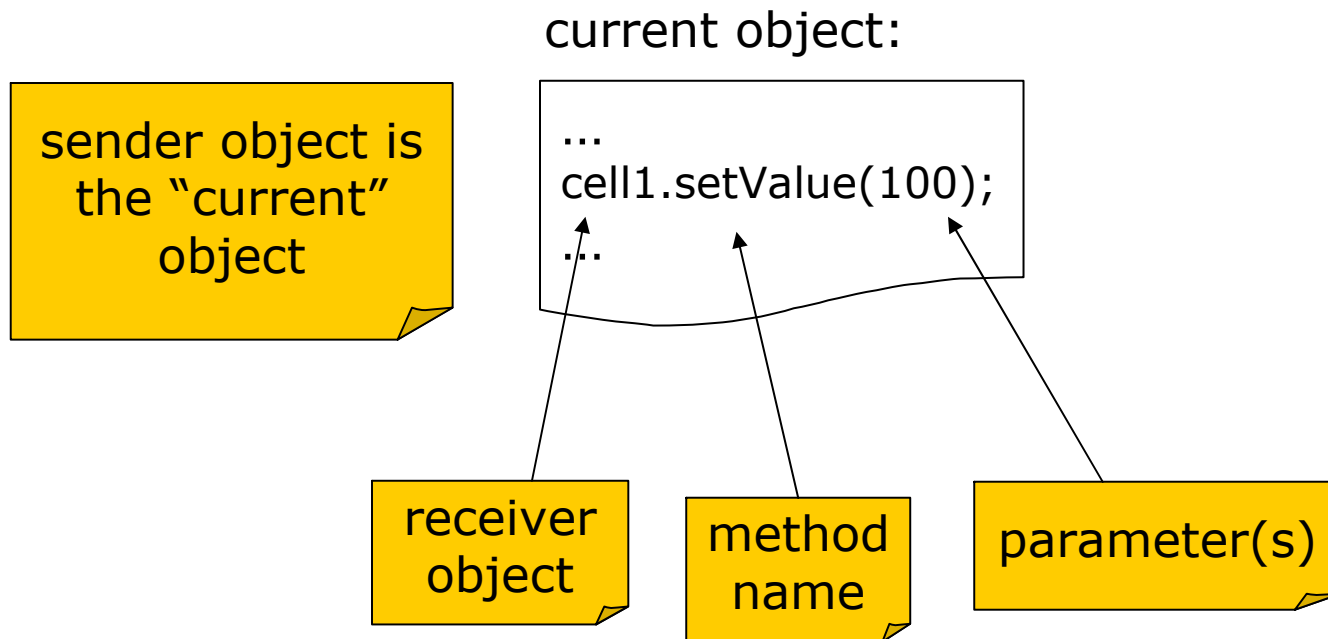
```
<returnType> <methodName> (<argument-list>)  
  {  
    <method-body>  
  }
```

A diagram illustrating the structure of a method signature and body. The text is: `<returnType> <methodName> (<argument-list>)` followed by a block `{ <method-body> }`. Red arrows point from the text above to the corresponding parts of the code: one arrow points from the first part of the text "(return type, method name, arguments)..." to `<returnType>`, another from `<methodName>`, a third from `<argument-list>`, and a fourth from the text "... and method body (the implementation)" to `<method-body>`.

- ... and method body (the implementation)

Message Sending

- Objects communicate by sending messages (i.e., by calling methods)



Constructor

- ❑ Piece of code that is executed to create an instance of a class and
- ❑ is used to initialize state of the instance.

Note: Make sure that you also call a constructor of the superclass (recursively) to initialize inherited data members!

Constructor Example

- Java does some things automatically

```
class Person {  
    String name;  
    Person(String paramName) {  
        name = paramName;  
    }  
}  
class Student extends Person {  
    int studentId;  
    Student(String paramName, int paramStudentId) {  
        super(paramName);  
        studentId = paramStudentId;  
    }  
}
```

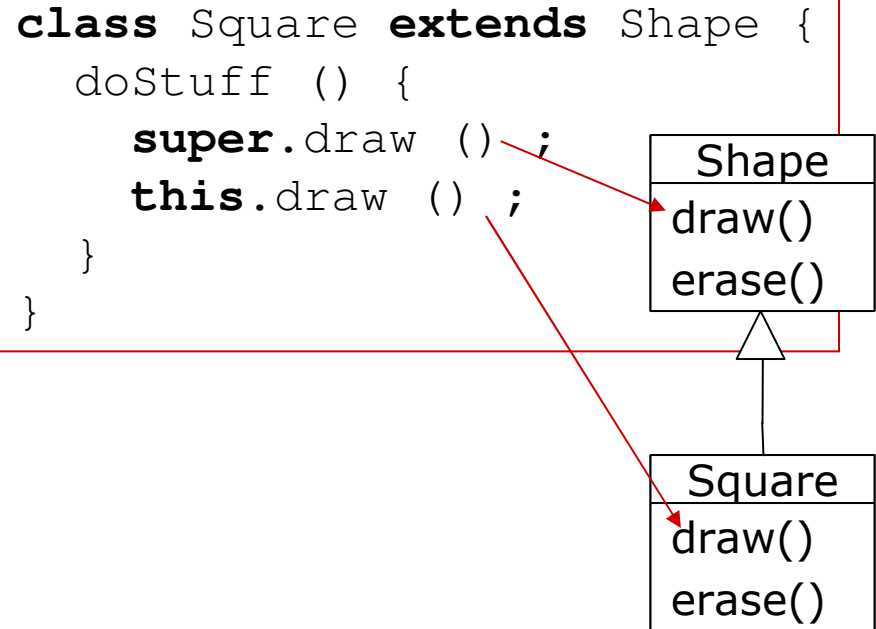
The statement `super(paramName)` calls the constructor `Person(String)` to initialize the inherited data member `name`.

References to “Current” Object

- Problem:
How can you write generic code referring to objects without having created them?
- Solution:
General name for a formal object of the current class (the “current” object) which is bound at runtime to an actual object.
 - Java calls this `this`
 - Some other languages `self`
- In addition there is a need for naming the super object of `this`, which Java calls `super`.

References to “current” object

- **this** vs. **super**
 - both references to the “current” object
- **this**
 - lookup of members begins at the object’s class
- **super**
 - lookup begins at the super-class of the object’s class
- **Note: no super.super!**



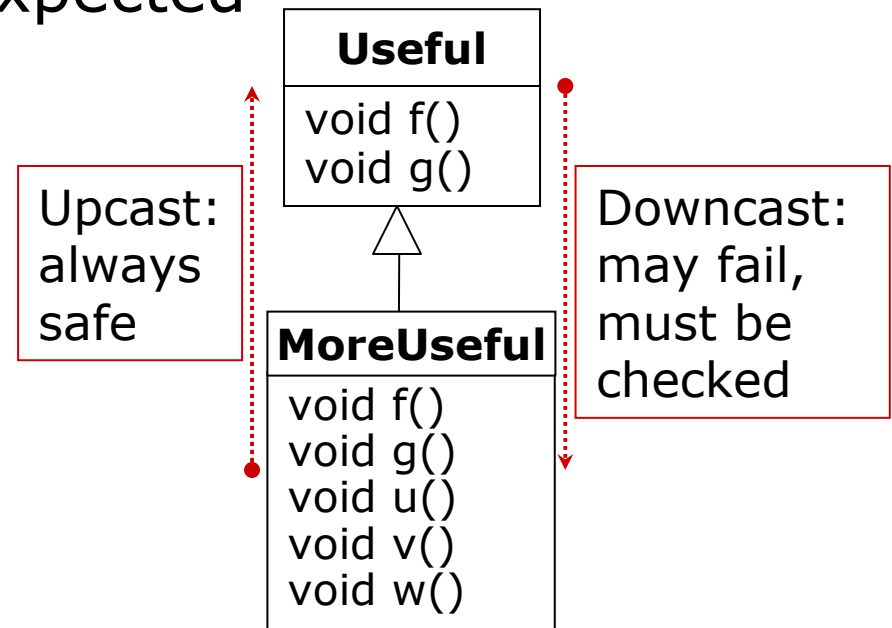
Visibility

- Protection of methods and attributes.
- Nothing new to object-orientation, but very useful for encapsulation.
- Example: visibilities in Java:

Modifier	Methods and Attributes are visible ...
public	from anywhere
protected	from subclasses and classes in the same package
<i>default</i>	from class in the same package
private	from the defining class

Casting

- Using a subclass object where a base class object is expected implies upcasting
- If you want to access methods of the extended interface you have to downcast the object



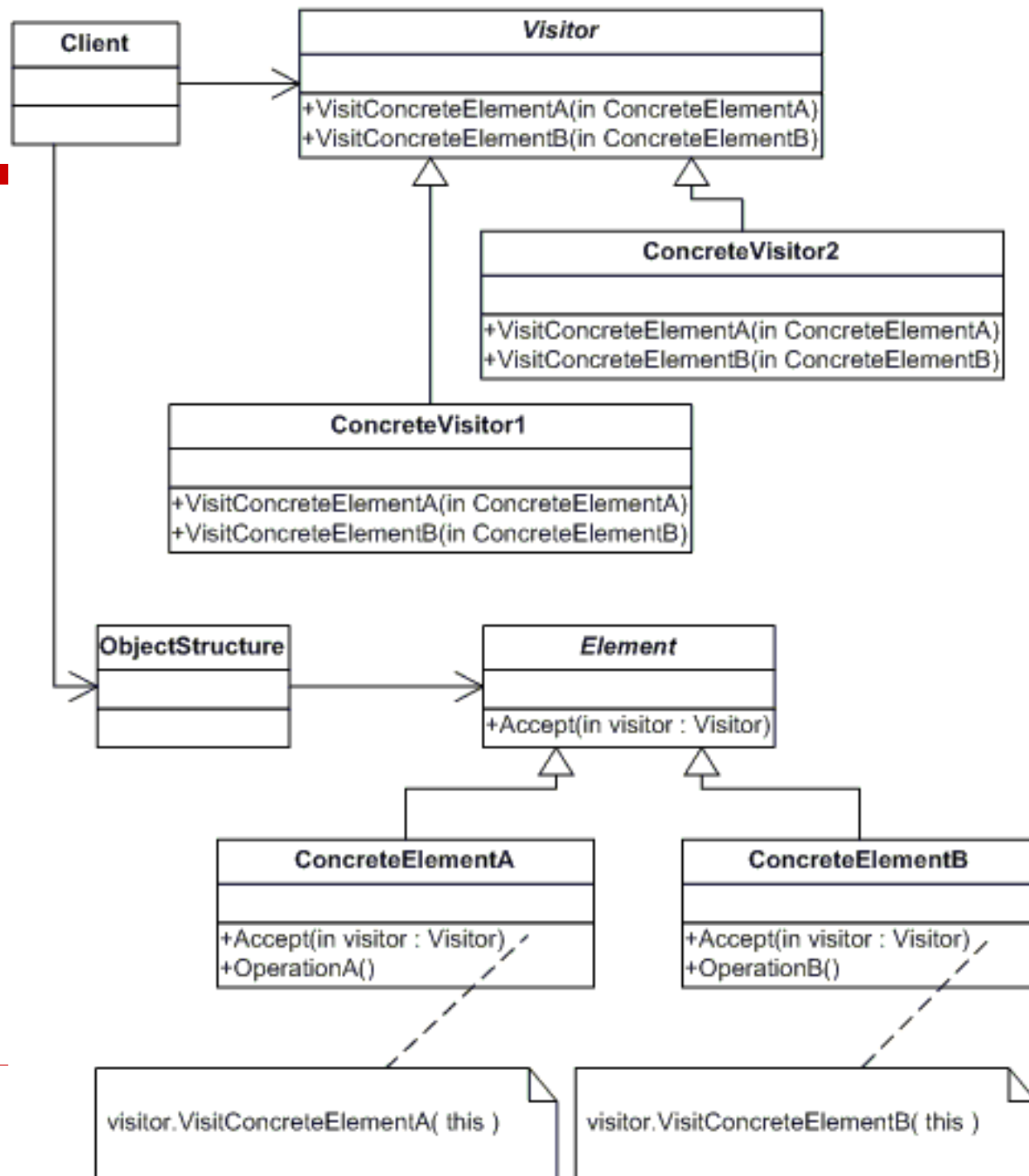
Downcast examples

```
Shape s = new Circle();
Circle c = (Circle) s; // ok: s happens to be a Circle
Rectangle r = (Rectangle) s; // runtime error
```

Runtime Type Identification

- ❑ Objects “know” their type at runtime.
- ❑ Runtime Type Identification (RTTI).
- ❑ E.g., you can at least detect type problems:

```
if (s instanceof Rectangle) {  
    Rectangle r = (Rectangle)s ;  
    ...  
}  
else // what to do???  
    throw new RuntimeException (  
        “unexpected type ”  
        + r.getClass ().getName ()) ;
```
- ❑ Using the Visitor pattern can avoid casts.

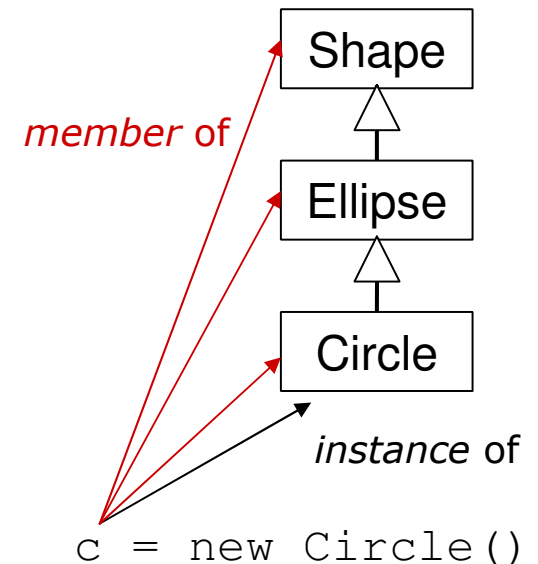


Object Orientation

More Concepts

Classes Revisited

- A class definition deals with the specification of:
 - structure (attributes)
 - behaviour (methods)
 - inheritance (parent relationship)
- Difference between
 - being a member of a class and
 - being an instance of a class
 - An object is an *instance* of its most specific class (called "its class"), but
 - is also a *member* of multiple classes (the ones its most specific class inherits from).



Abstract Classes

- ❑ Abstract classes cannot be instantiated.
- ❑ Can contain abstract methods, i.e., have no method body.
- ❑ Abstract classes have to be overridden by subclasses which then can be instantiated.
- ❑ Abstract classes can contain anything that is possible in normal classes.

Interfaces

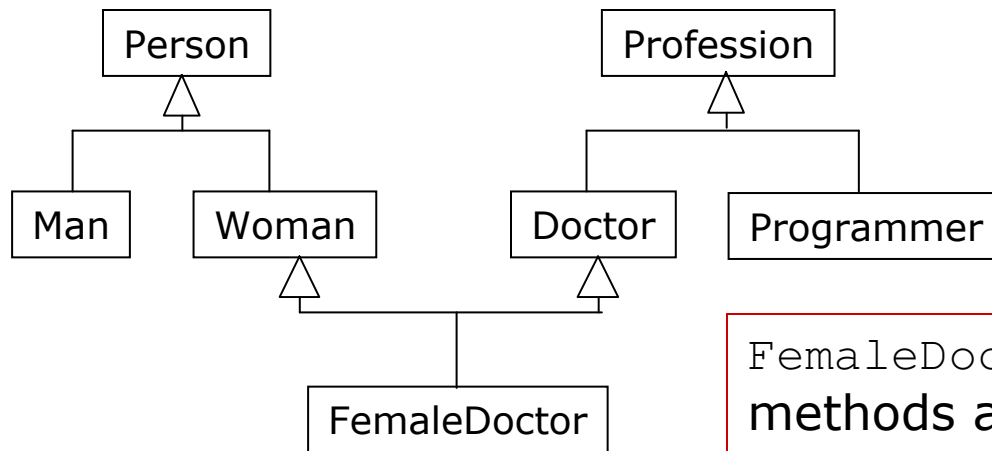
- ❑ Cannot contain any implementation.
- ❑ Classes implement interfaces by giving bodies for the required methods.
- ❑ “private” visibility does not make sense.

```
interface List {  
    public void add(Object o);  
    public Object get(int index);  
}
```

```
class ArrayList implements List {  
    private Object[] data ;  
    public void add (Object o) {  
        /* Implementation */  
    }  
    public Object get (int index) {  
        /* Implementation */  
    }  
}
```

Multiple Inheritance

- Class inherits from multiple parents



`FemaleDoctor` now has all the methods and attributes of both `Doctor` and `Woman` (and all their superclasses).

- Can cause problems with duplicate names
 - Need some means to resolve conflicts
- Not available in Java, but, e.g., in C++

Forms of Typing

- Static
 - type information available at compiletime only
 - static typing can be checked by the compiler
- Dynamic
 - type information also available at runtime
 - uses inherent types of polymorphic objects
 - this cannot be checked by the compiler but only during execution

Garbage Collection

- Garbage Collection
 - Automatically disposes objects that are no longer reachable
 - Avoids bugs due to missing deallocation
- Explicit destruction
 - It is sometimes difficult to know when to destroy.
 - Can break encapsulation, b/c you need knowledge of the whole program to determine when to destroy

Metaclass

- A class' class
 - if a class is represented by an object this object's class is called "metaclass"
 - think of how a compiler would store information about the class you defined
- Metaclasses can be made accessible to programs
 - You get all sorts of information about the original class (attributes, methods, visibility, inheritance, ...)
- This is not a necessity for OO, some languages, e.g., C++ don't have it.
- Other languages also have a class' class' class (e.g. Smalltalk).

Metaclass Example

- ❑ For details on Java, see `java.lang.Class` and package `java.lang.reflect.*`
- ❑ Java metaclass mechanism is called reflection. Reflection is read-only.

Java Example

```
void inform (Object o) {  
    Class oClass = o.getClass () ;  
    System.out.println ("Classname: " + oClass.getName ()) ;  
    Method [] methods = oClass.getMethods () ;  
    for (int i = 0; i < methods.length; i++)  
        System.out.println ("method: " + methods [i].getName ()) ;  
}
```

(Note: You can also add class members, this is called code-injection.)

Java's Metaclass Replacements

- ❑ Java lacks “real” metaclasses.
- ❑ Instead: built-in concepts.
- ❑ Modifier **static** for class members:

```
abstract class WebPage {  
    static private int numberOfAccesses;  
    static public getNumberOfAccesses () {  
        return numberOfAccesses ;  
    }  
    ...  
} // class WebPage
```
- ❑ Modifiers **abstract** and **interface** for classes.

Levels of Object-Orientation

Level 1: only objects, these can serve as “prototypes” (prototype-based languages)

Level 2: objects and classes but no retrospection on classes (e.g. C++)

Level 3: objects, classes and meta-classes (e.g. Java)

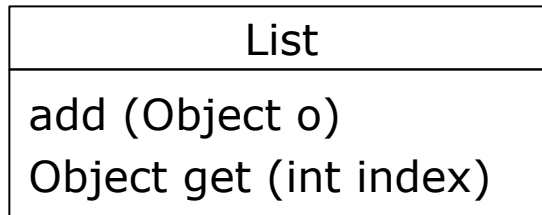
(Level 4: object, class, class class, meta-class meta-class class, Smalltalk)

Generics

- Parameterize classes and methods with classes
 - Example: List of String, List of Person
- Allows you to implement a class that deals with objects of another type.
- Makes the language more type-safe (also known as

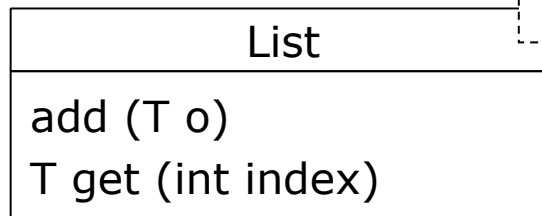
parametric polymorphism)

language
without generics:

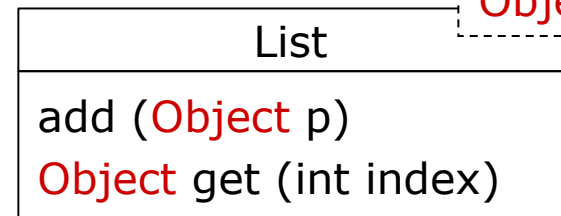


language
with generics:

generalizes to:



then equivalent to:



T

Object

Generics in Java

```
class List<E> {  
    void add(E data);  
    E get(int index);  
    /* ... */  
};
```

□ Without generics:

```
List dogList = new ArrayList();  
dogList.add(new Dog());  
dogList.add(new Cat()); //this compiles and runs fine
```

□ With generics:

```
List<Dog> dogList = new ArrayList<Dog>();  
dogList.add(new Dog());  
dogList.add(new Cat()); // this is a compile-time error
```

□ New in Java 5.

□ Java 5: Type erasure

- Substitution of non-generic classes and casts
- Generics information not present at runtime
- Implicit casts needed for arguments and return values

Generics in C++

- Generics are implemented as Templates
 - New classes created for every use of the generic class
 - Can blow up code considerably

```
template<class T>
class List {
    public:
        void add (T data) ;
        T get (int index) ;
        List () { /* ... */ }
    private:
        // ... the data ...
};
```

Generics with Wildcards

- Generics allow to specify that a type is not known (wildcards)

```
List<?> list = new ArrayList<String> () ;  
list.add (new Object ()) ; // does this work?  
Object o = list.get (0) ; // what about this?
```

- Constrained Wildcards (bounded polymorphism)
 - “Any list that is a list of something that is a member of Shape”

```
void draw (List<? extends Shape> shapes) {  
    for (Shape s : shapes) s.draw();  
}
```

- **Also:** ? super Shape.

Here you know that everything in `shapes` is a shape, because `shapes` is a list of one of the subtypes of `Shape` (including `Shape`).

Generics' Catches

- Be careful about what the bounding means

```
void addCircle (List<? extends Shape> shapes) {  
    shapes.add (new Circle ()); //Why does this not work?  
}
```

shapes could e.g. be a List<Rectangle>

Instead, use:

```
void addCircle (List<Shape> shapes) {  
    shapes.add (new Circle ()); ; //now this works  
}
```

shapes **is** a List<Shape>. Since Circle **certainly is** a Shape, this works.

Covariance

- Type relation of `List<Shape>` and `List<Circle>`?
 - No !!
- But Java arrays have the property that their types are *covariant*
 - `String[]` subtype of `Object[]`

On OO: Links and References

Links to OO:

- Object FAQ - <http://www.ipipan.gda.pl/~marek/objects/faq/>
- Don't fear the OOP (OO & Java Tutorial) - <http://sepwww.stanford.edu/sep/josman/oop/oop1.htm>
- Sun's Java 5 page - <http://java.sun.com/j2se/1.5.0/>

References for Chap.1:

- Abadi, Cardelli: A Theory of Objects - Springer Monographs in Computer Science
- Alan Kay (five rules)