

Principles of Object-Orientation

- Encapsulation
- Inheritance
- Polymorphism

CE202 Software Engineering, Autumn term

Dr Michael Gardner, School of Computer Science and Electronic Engineering, University of Essex



Object-orientation

- ▶ So far we have briefly touched on how analysis and design (and implementation) can make use of objects
 - ▶ Eg. objects in activity diagrams, class diagrams, etc
- ▶ In this lecture we will explore common object-oriented principles
- ▶ As we get more into UML we will make more use of objects



Advantages of O-O

- ▶ Can save effort
 - ▶ Reuse of generalized components cuts work, cost and time
- ▶ Can improve software quality
 - ▶ Encapsulation increases modularity
 - ▶ Sub-systems less coupled to each other
 - ▶ Better translations between analysis and design models and working code
 - ▶ Objects are good for modelling what happens in the real world
 - ▶ Can be used throughout the software lifecycle ie.
requirements -> design -> implementation -> testing



OO Analysis & Design: mechanisms of abstraction

- ▶ Fundamentally: the same abstraction mechanisms as object-oriented programming:
 - ▶ Encapsulation: classes and objects
 - ▶ Other possible modularization techniques: interfaces, packages/namespaces
 - ▶ Inheritance
 - ▶ Generalization/specialization
 - ▶ Subtyping
 - ▶ Subclassing
 - ▶ Polymorphism
 - ▶ Overloading
 - ▶ Dynamic binding



Encapsulation

Objects

An object is:

“an abstraction of something in a problem domain, reflecting the capabilities of the system to

- ▶ keep information about it,
- ▶ interact with it,
- ▶ or both.”

Coad and Yourdon (1990)

Objects

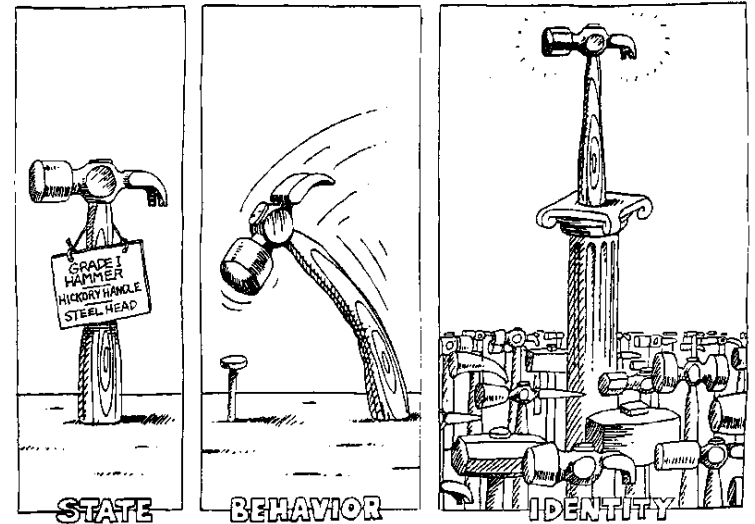
“Objects have state, behaviour and identity.”

Booch (1994)

- ▶ *State*: the condition of an object at any moment, affecting how it can behave
- ▶ *Behaviour*: what an object can do, how it can respond to events and stimuli
- ▶ *Identity*: each object is unique

Object: Definition

- ▶ In OOP: Primary mean of abstraction
- ▶ An object is defined by:
 - ▶ State: Attributes (data)
 - ▶ Behaviour: operations
 - ▶ Identity
 - ▶ does not depend on the current value of the attributes
 - ▶ never changes
- ▶ → Each object has at each point in time—
 - ▶ state: the current value of the attributes
 - ▶ behaviour: the set of operations they recognize, and the way they are interpreted
 - ▶ identity



Examples of Objects

| Object | Identity | Behaviour | State |
|----------------------------|---|-----------------------------|--------------------------------------|
| A person | 'Hussain Pervez.' | Speak, walk, read. | Studying, resting, qualified. |
| A shirt | My favourite button white denim shirt. | Shrink, stain, rip. | Pressed, dirty, worn. |
| A sale | Sale no #0015, 18/05/05. | Earn loyalty points. | Invoiced, cancelled. |
| A bottle of ketchup | <i>This</i> bottle of ketchup. | Spill in transit. | Unsold, opened, empty. |

Can you suggest other behaviours and states for these objects?

How can external events and object behaviour both result in a change of state?

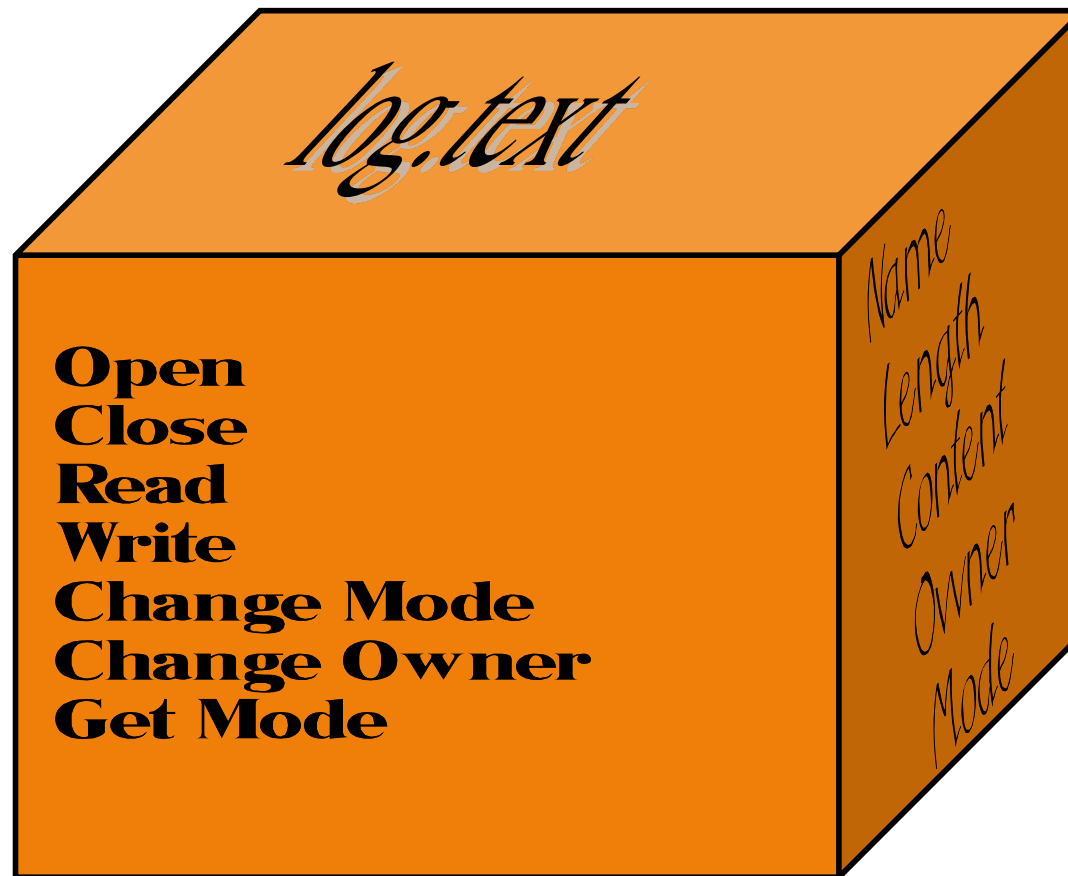
How can state restrict the possible behaviours of an object?

Examples of objects (cont.)

▶ Examples:

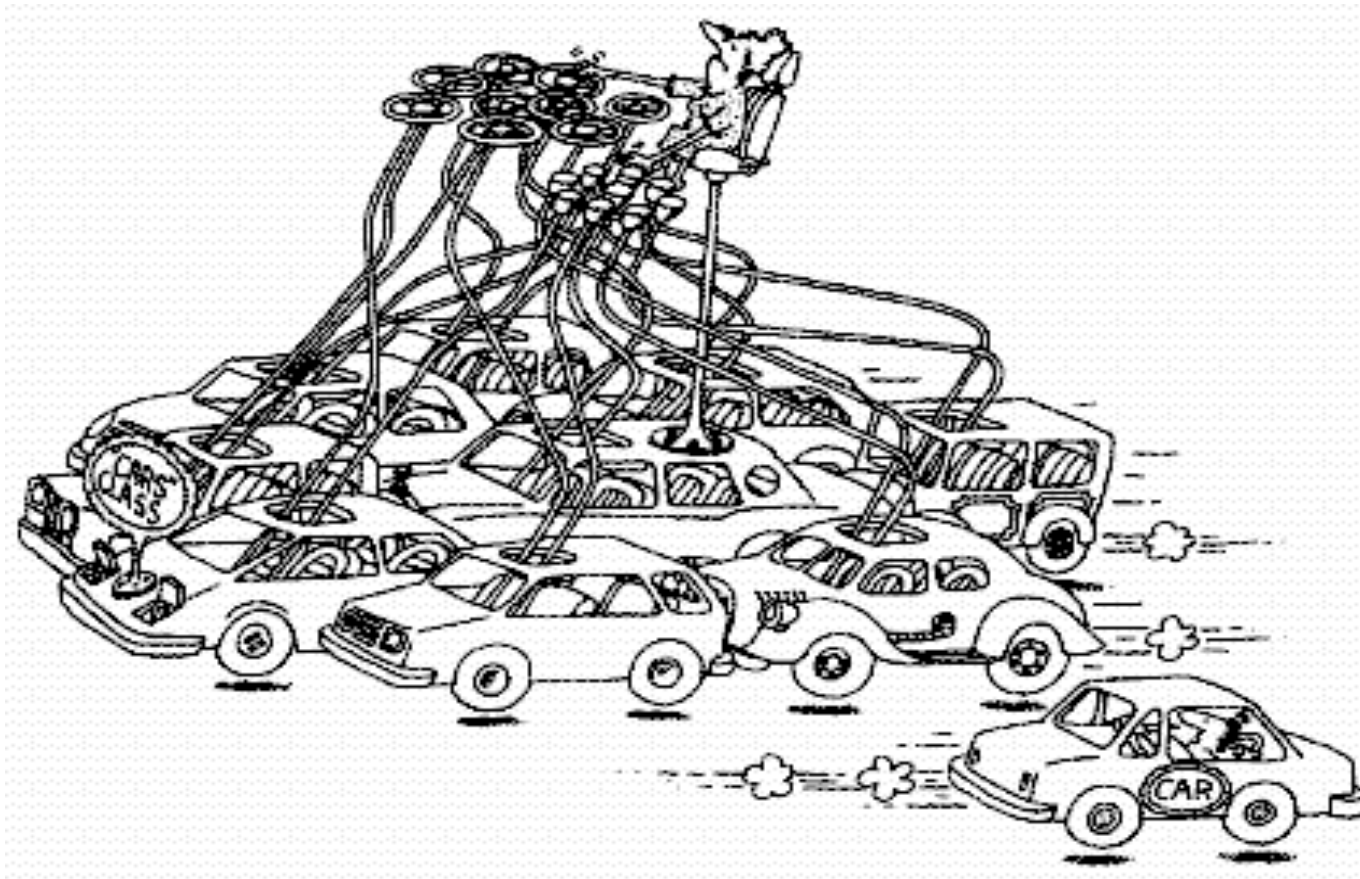
- ▶ Time point
 - ▶ Data: 16:45:00, Feb. 21, 1997
 - ▶ Operations: add time interval, calculate difference from another time point,
- ▶ Acts. For example: Measurement of a patients' fever
 - ▶ Data: 37.1C, by Deborah, at 10:10 am
 - ▶ Operations: Print, update, archive
- ▶ File
 - ▶ Data: log.txt, -rwx-----, Last read 21:07 June 1, 1999, ...
 - ▶ Operations: read, write, execute, remove, change directory, ...
- ▶ A communication event (time, length, phone-number, ...)
- ▶ Transaction in a bank account (withdraw \$15, time, ...)
- ▶ Elements of ticket machine dispenser: ticket, balance, zone, price, ...

Object: Example



Class: Abstraction Over Objects

- ▶ A class represents a set of objects that share a common structure and a common behavior.



Class and Instance

- ▶ All objects are *instances* of some *class*
- ▶ A Class is a description of a set of objects with similar:
 - ▶ features (attributes, operations, links);
 - ▶ semantics;
 - ▶ constraints (e.g. when and whether an object can be instantiated).

OMG (2009)



Class and Instance

- ▶ An object is an instance of some class
- ▶ So, instance = object
 - ▶ but also carries connotations of the class to which the object belongs
- ▶ Instances of a class are similar in their:
 - ▶ *Structure*: what they *know*, what information they hold, what links they have to other objects
 - ▶ *Behaviour*: what they *can do*



What is a Class?

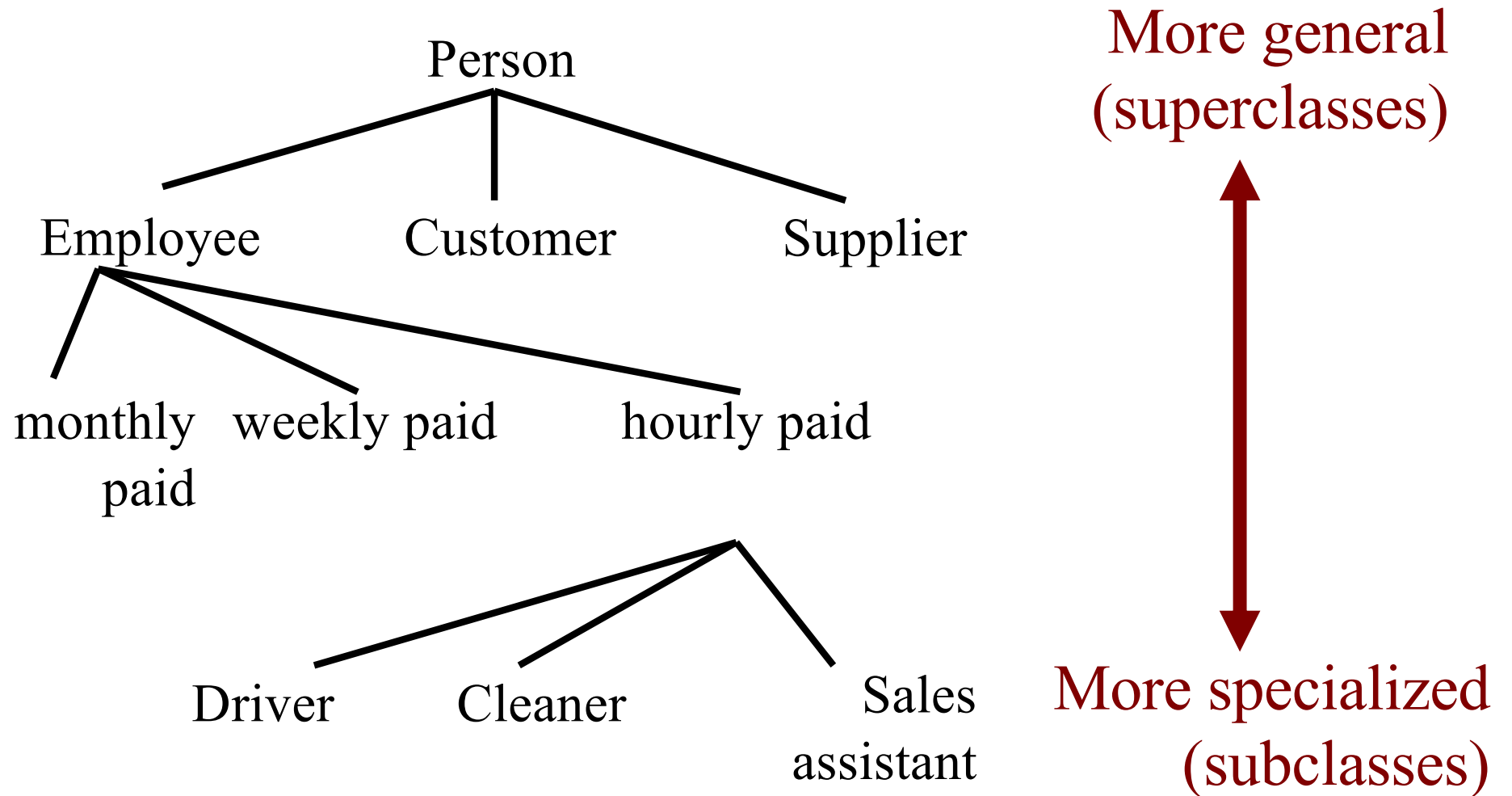
- ▶ Stands for a single human concept
 - ▶ An abstraction technique
- ▶ Mould for objects
 - ▶ used to instantiate objects (instances) with distinct identities that share protocol, behavior and structure but may assume different states.
 - ▶ In contrast to concrete object, a class does not necessarily exist in (run) time and (memory) space.
- ▶ What's not a Class?
 - ▶ An object is not a class.
 - ▶ ... but a class may be an object:
 - ▶ In "exemplar based" languages eg. Smalltalk, there are no classes. New objects are "instantiated" from existing objects.

Generalization and Specialization

- ▶ Classification is hierarchic in nature
- ▶ For example, a person may be an employee, a customer, a supplier of a service
- ▶ An employee may be paid monthly, weekly or hourly
- ▶ An hourly paid employee may be a driver, a cleaner, a sales assistant



Specialization Hierarchy



Generalization and Specialization

- ▶ More general bits of description are *abstracted out* from specialized classes:

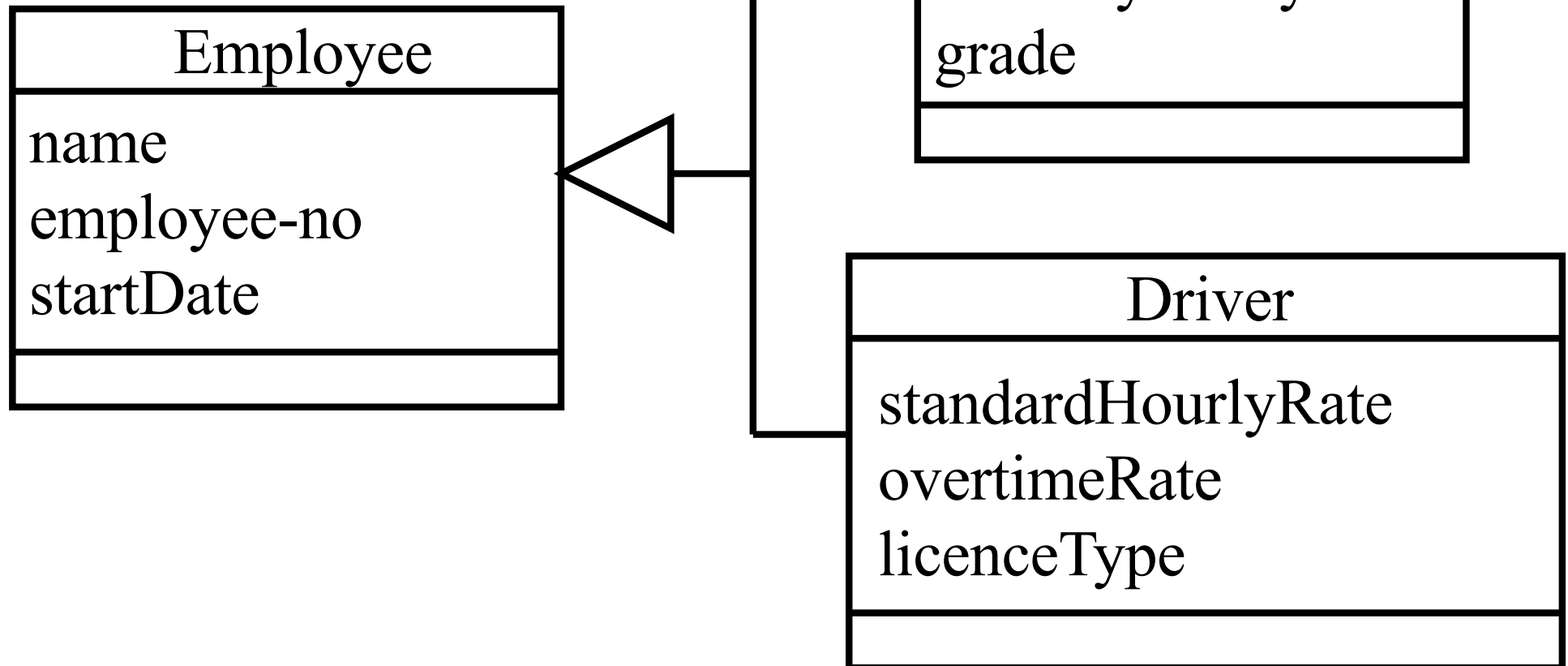
| SystemsAnalyst |
|--|
| name employee-no startDate monthlySalary grade |
| |

| Driver |
|---|
| name employee-no startDate standardHourlyRate overtimeRate licenceType |
| |

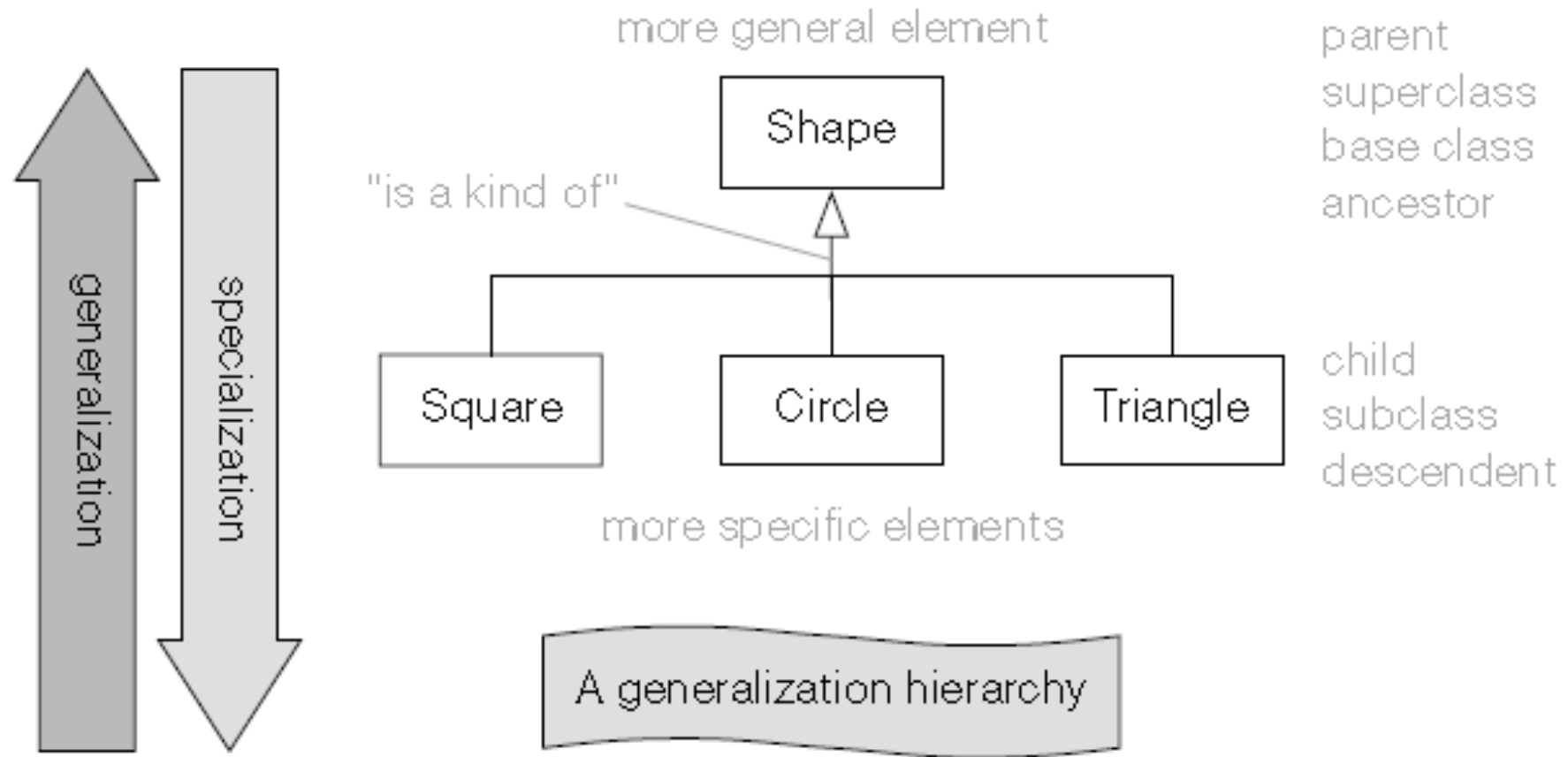


Specialized (subclasses)

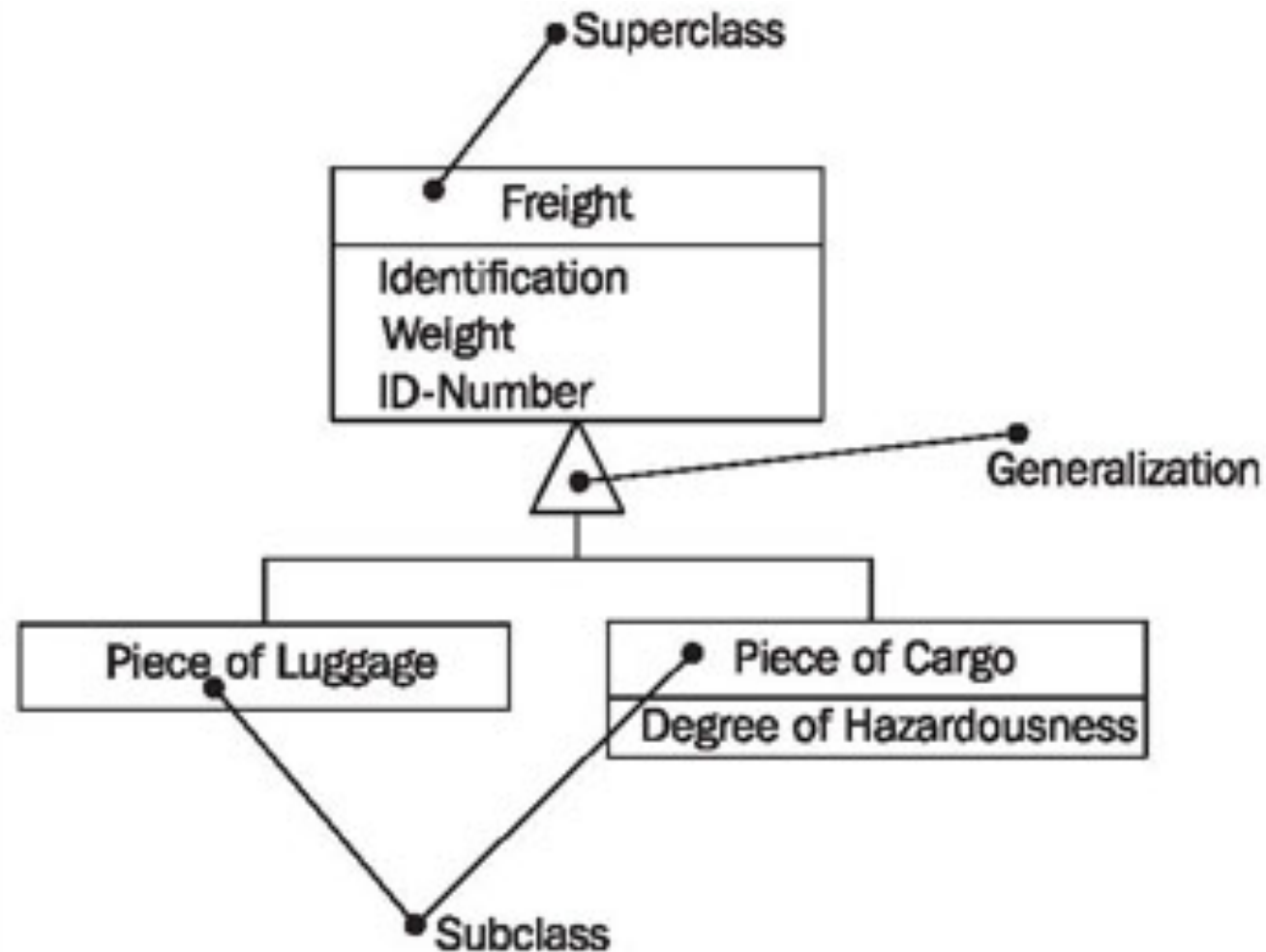
General (superclass)



Generalization

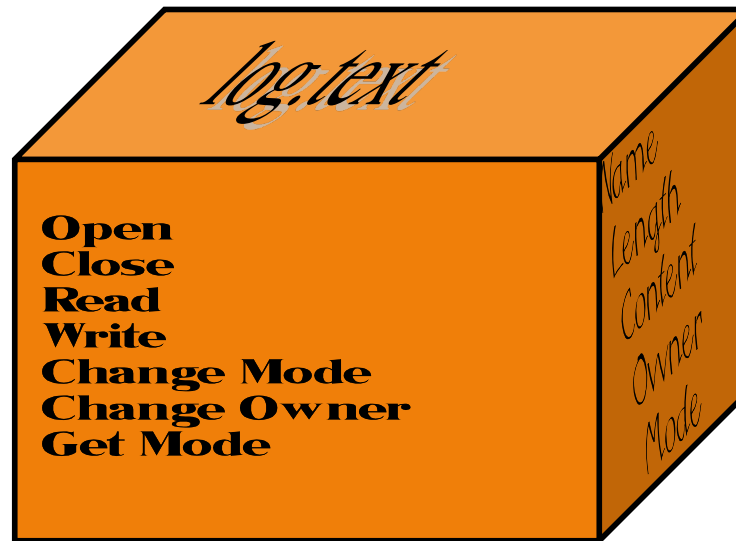


Object-oriented modelling notation: genericity in class diagrams



Encapsulation definition

- ▶ Encapsulation: an object's data is located with the operations that use it



Message-passing

- ▶ Several objects may collaborate to fulfil each system action/use-case
- ▶ “Record CD sale” could involve:
 - ▶ A CD stock item object
 - ▶ A sales transaction object
 - ▶ A sales assistant object
- ▶ These objects communicate by sending each other messages



Message-passing and Encapsulation

‘Layers of an onion’
model of an object:

An outer layer of
operation signatures...

...gives access to middle
layer of operations...

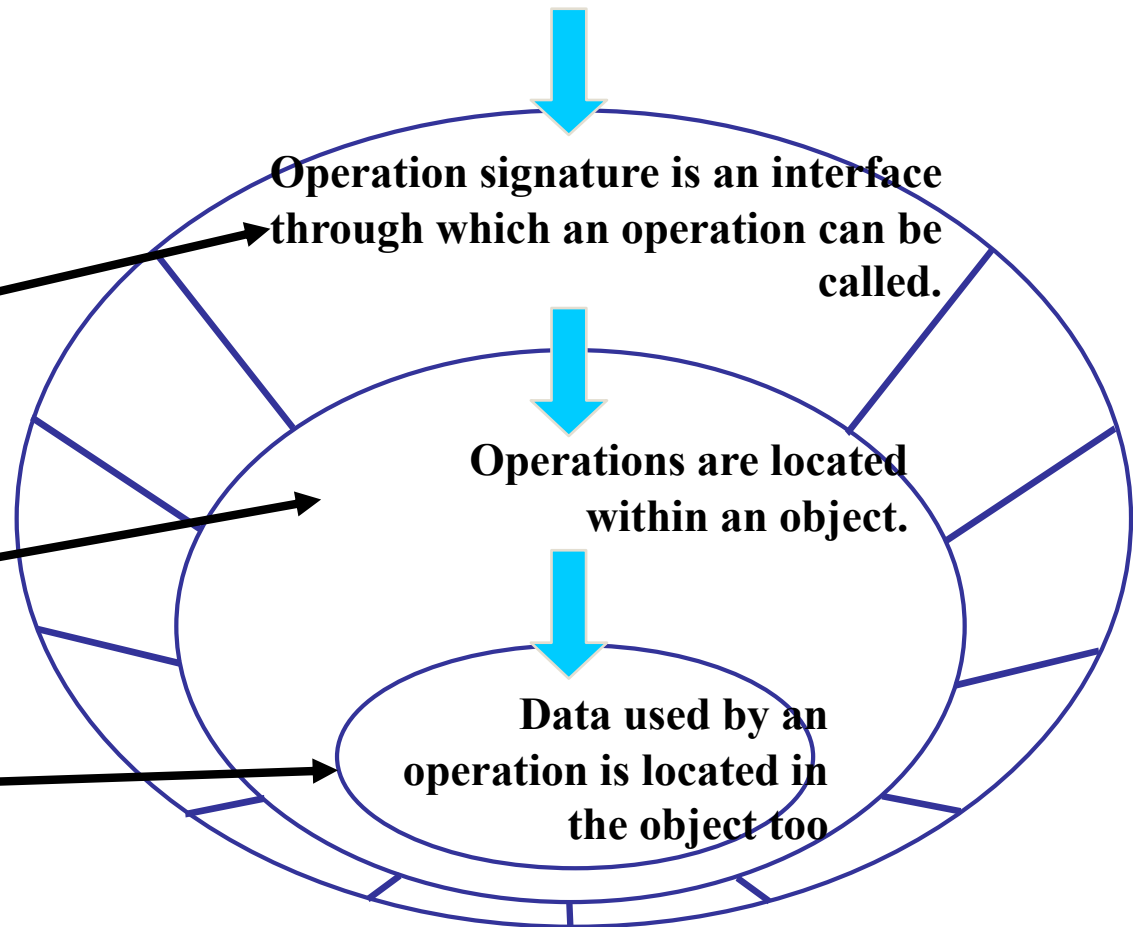
...which access an
inner core of data

Message from another object
requests a service.

Operation signature is an interface
through which an operation can be
called.

Operations are located
within an object.

Data used by an
operation is located in
the object too



Classes

- ▶ Kinds of classes:
 - ▶ Abstract (also: Java interface)
 - ▶ No instances!
 - ▶ Serve as an interface, or a common base with similar behavior
 - ▶ Example: `Collection` (Smalltalk, Java)
 - ▶ Singleton
 - ▶ One instance!
 - ▶ Example: `True` (Smalltalk)
 - ▶ Language support for singletons:
 - Self: No classes! All objects are singletons
 - BETA: Objects may be defined as singletons
 - ▶ Concrete/effective
 - ▶ Any number of instances

HOW TO SLAUGHTER A PIG



PHASE 1: TRY IT WITHOUT -9

Classes in O-O programming languages

▶ C++

```
class Employee: public Person {  
private:                        // visible to none  
    Date birthday;             // data member  
public:                        // visible to all  
    void hire(Reason why)      // function member  
    { /* ... */ }  
};
```

▶ Java:

```
class Employee extends Person {  
    private Date birthday;      // field, visible to none  
    public void Hire(Reason why) { // method, visible to all  
        { /* ... */ }  
    }  
}
```

Classes in OOPs (Cont.)

▶ Smalltalk:

```
Person subclass: Employee
  instanceVariables: 'birthday' "instance variable, visible to none"
  classVariables: ' '
  poolDictionaries: ' '
```

▶ Eiffel:

```
hire: why "method, visible to all"
"..."
```

```
class EMPLOYEE
inherit
  PERSON
feature {NONE} -- visible to none
  DATE birthday; -- attribute
feature {ALL} -- visible to all
  hire(why: REASON) is -- routine
  do
    -- ...
  end
end -- class EMPLOYEE
```



Information Hiding

Information hiding

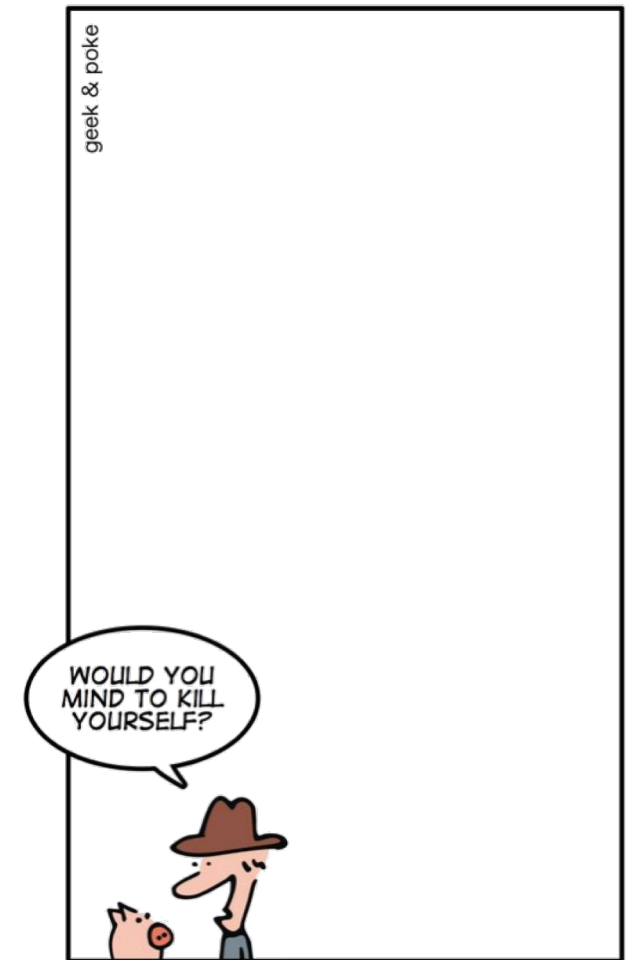
- ▶ Also known As: Data Abstraction, Data Hiding

... like us, objects have a private side. The private side of an object is how it does these things, and it can do them in any way required. How it performs the operations or computes the information is not a concern of other parts of the system. This principle is known as information hiding [Wirfs-Brock 1990, p.6].

- ▶ Why?

“It is an attempt to minimize the expected cost of software and requires that the designer estimate the likelihood of changes.” [Parnas 1985]

HOW TO SLAUGHTER A PIG



PHASE 1: TRY IT WITHOUT -9

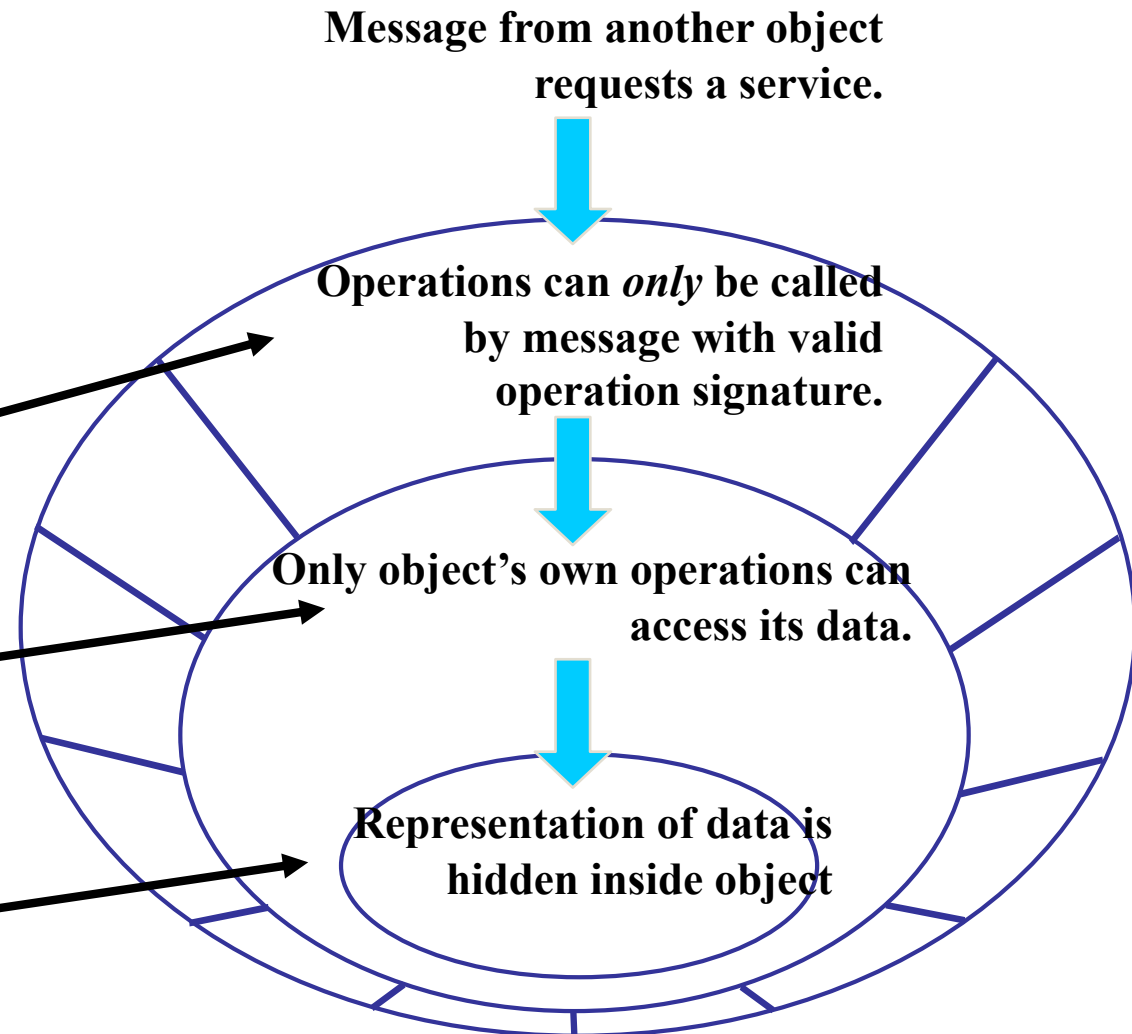
Information Hiding: the onion model

‘Layers of an onion’
model of an object:

Only the outer layer is
visible to other objects...

...and it is the only way to
access operations...

...which are the only way
to access the hidden data



Note:

Information Hiding Vs. Encapsulation

- ▶ The terms are sometimes confused and used interchangeably
 - ▶ Some people say “encapsulation” with reference to what we described as information hiding
- ▶ We shall adhere to the interpretation in these slides
 - ▶ Encapsulation: an object’s data is located with the operations that use it
 - ▶ **Information hiding: only an object’s Interface is visible to other objects**

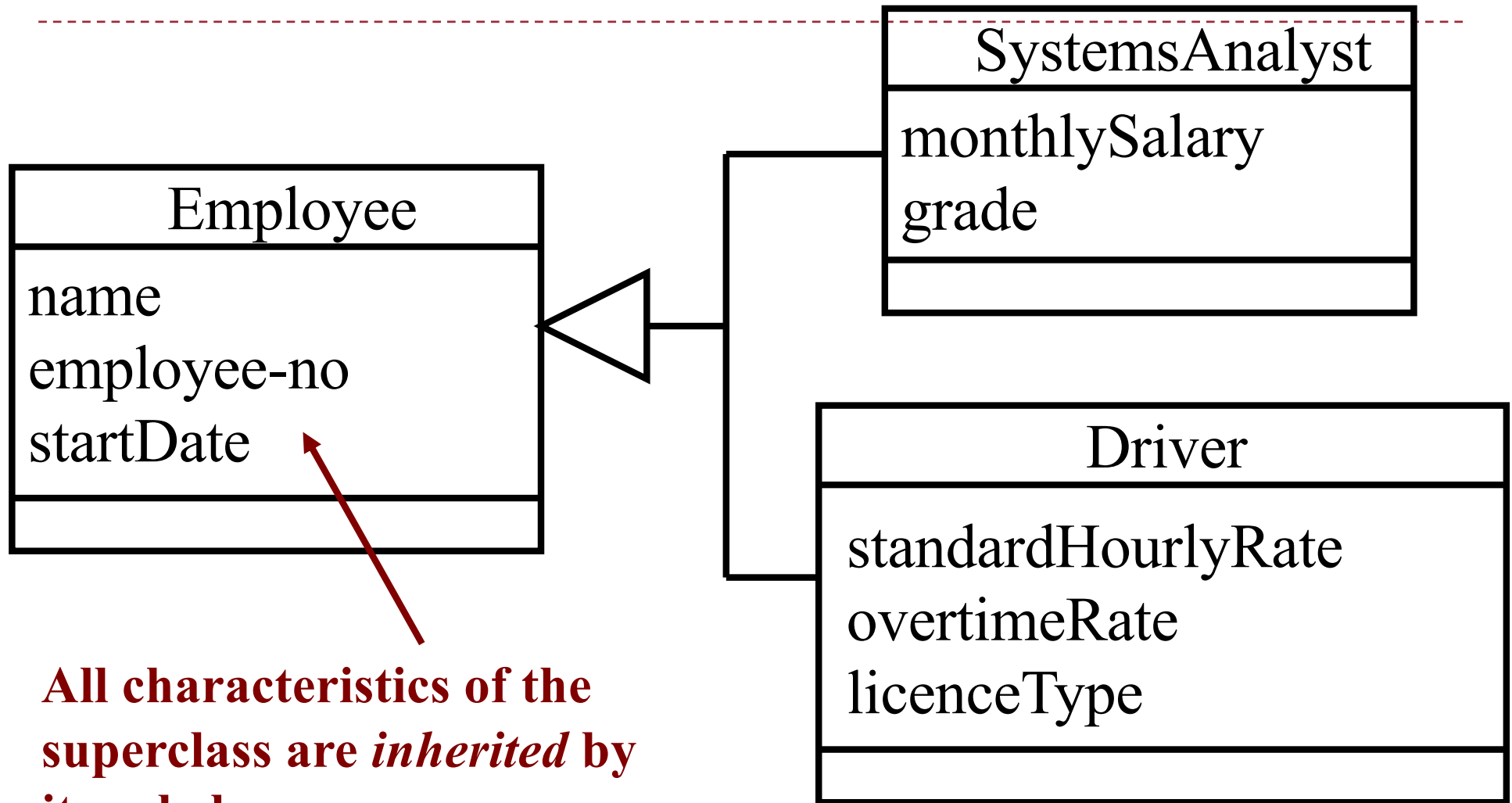


Inheritance

Inheritance

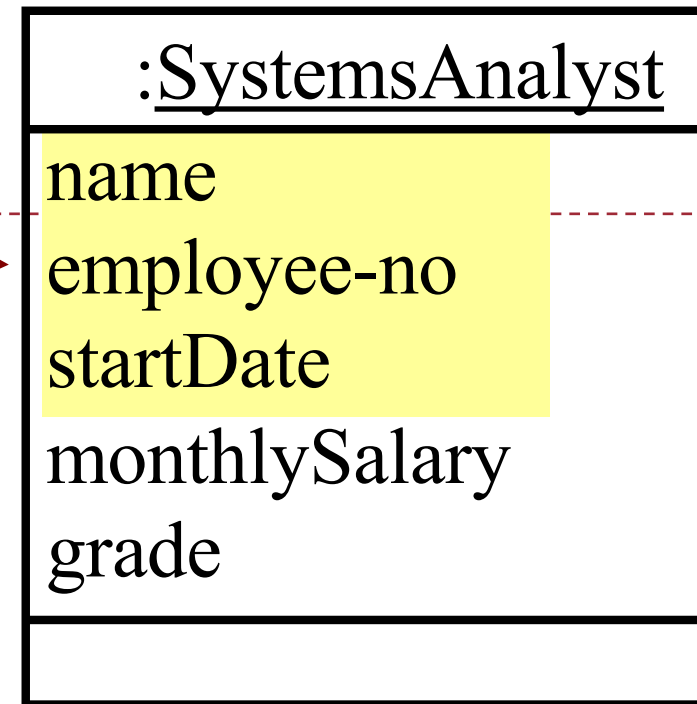
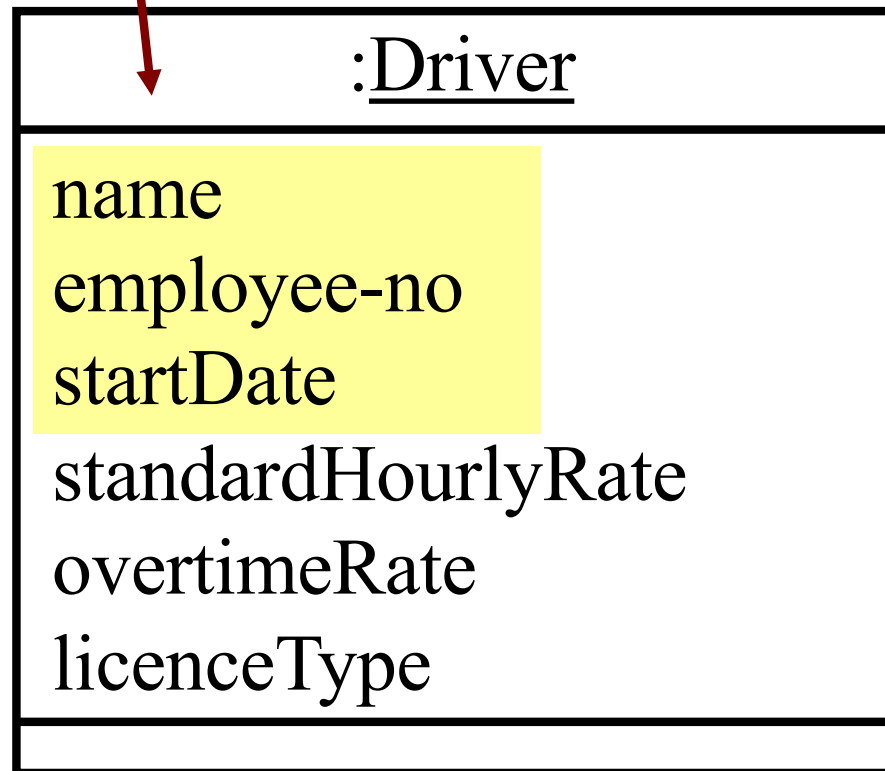
- ▶ The *whole* description of a superclass applies to *all* its subclasses, including:
 - ▶ Information structure (including associations)
 - ▶ Behaviour
- ▶ Often known loosely as *inheritance*
- ▶ (But actually inheritance is how an O-O programming language *implements* generalization / specialization)





All characteristics of the superclass are *inherited* by its subclasses

Instances of each subclass include the characteristics of the superclass (but not usually shown like this on diagrams)

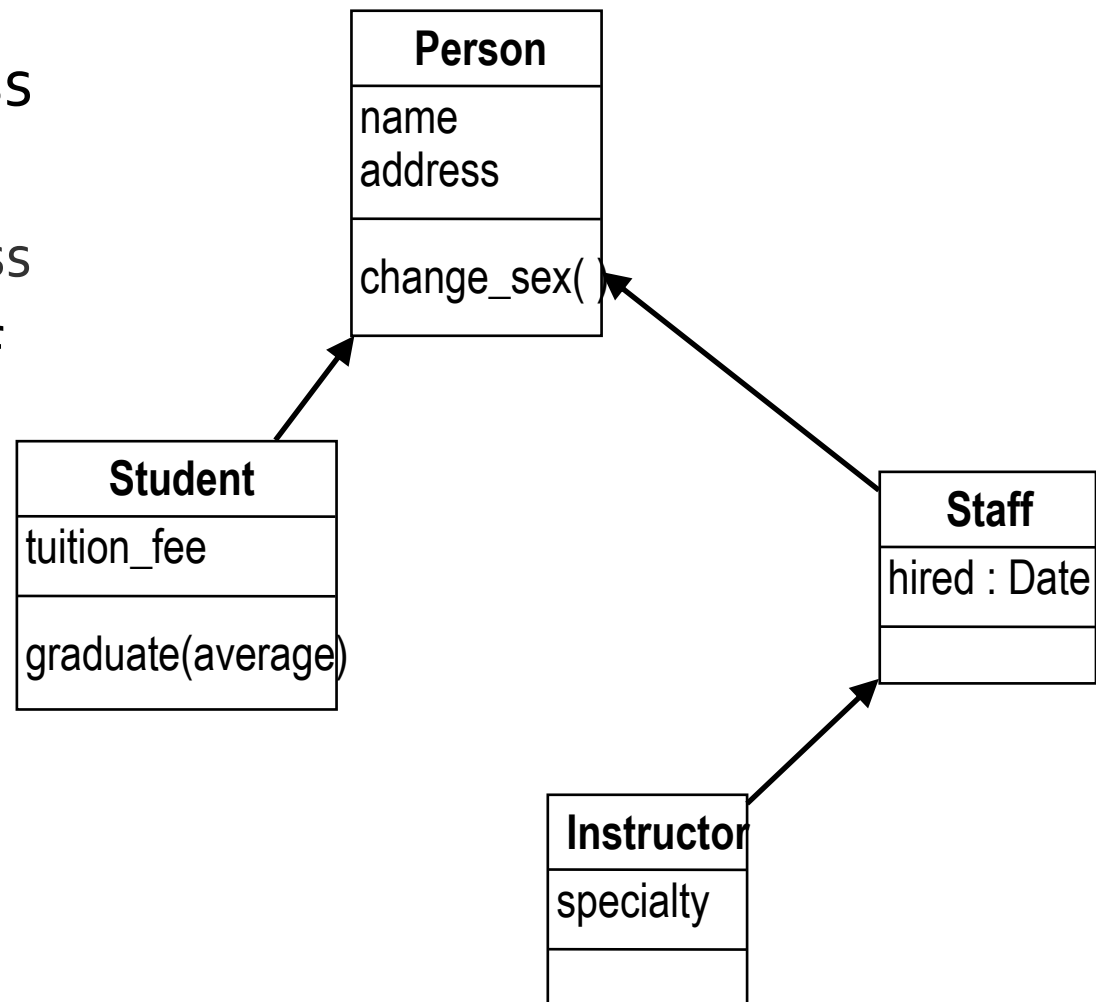


Inheritance

- ▶ Represents separate notions:
 - ▶ Generalization: ‘is-kind-of’ relation
 - ▶ Instances of the specialized class are a subcategory of the generalized class
 - ▶ Subtyping (in Java: ‘implements’)
 - ▶ The subtypes supports all the operations on supertype
 - ▶ Subclassing (in Java: extends)
 - ▶ A mechanism of code reuse

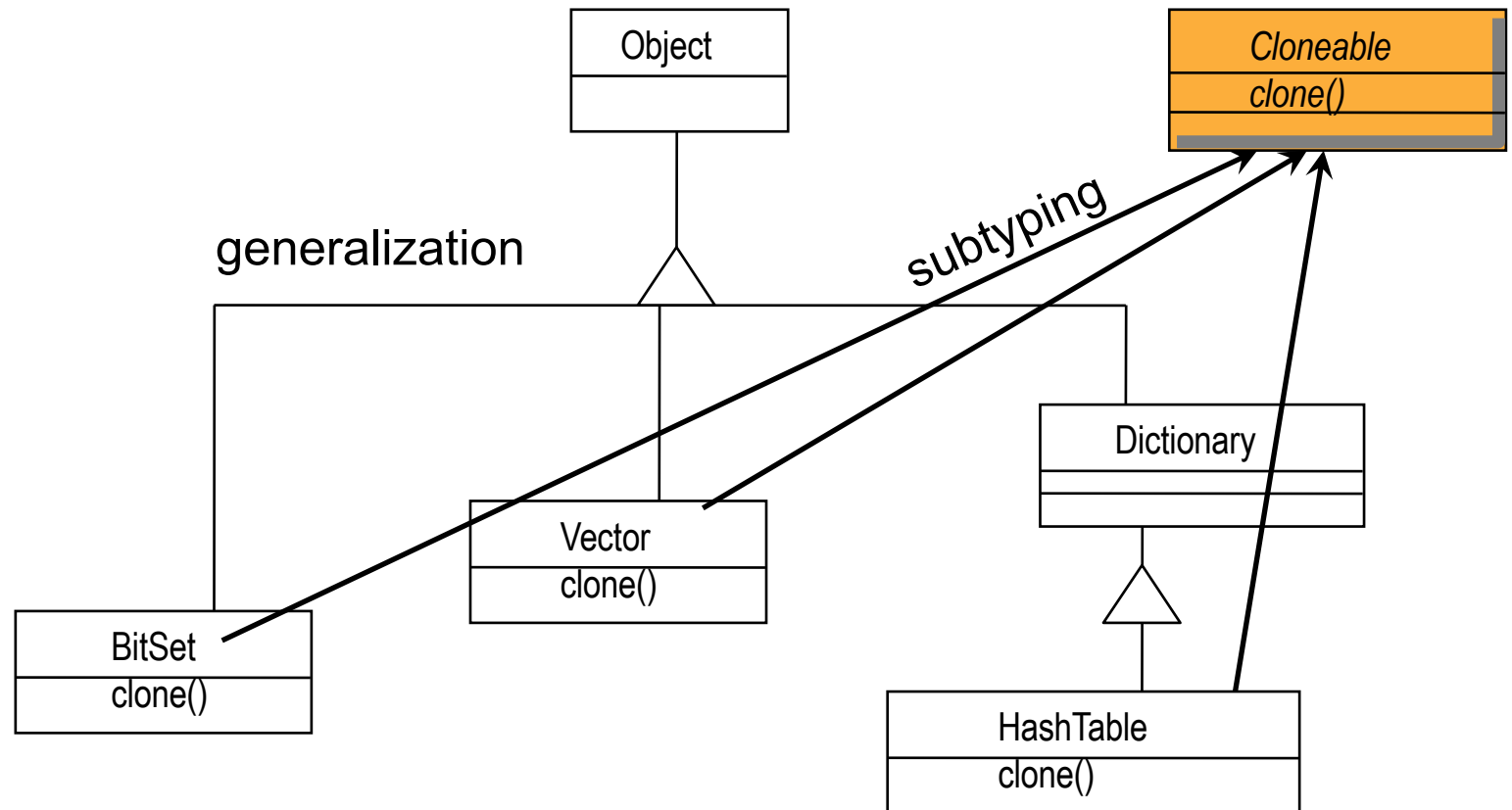
Inheritance *and* generalization

- ▶ Superclass is a generalization of Subclass
 - ▶ Also: subclass is a specialization of superclass
- ▶ Also known as: is-kind-of relation:
 - ▶ Staff is-kind-of Person
 - ▶ Instructor is-kind-of Person



Inheritance *and* subtyping

- ▶ Subtype supports the same operations as supertype



Subtyping enables a given type to be substituted for another type or abstraction. Subtyping is said to establish an **is-a** relationship between the subtype and some existing abstraction, either implicitly or explicitly, depending on language support. The relationship can be expressed explicitly via inheritance in languages that support inheritance as a subtyping mechanism.

In some OOP languages, the notions of code reuse and subtyping coincide because the only way to declare a subtype is to define a new class that inherits the implementation of another.

A class implements the Cloneable interface to indicate to the Object.clone() method that it is legal for that method to make a field-for-field copy of instances of that class.

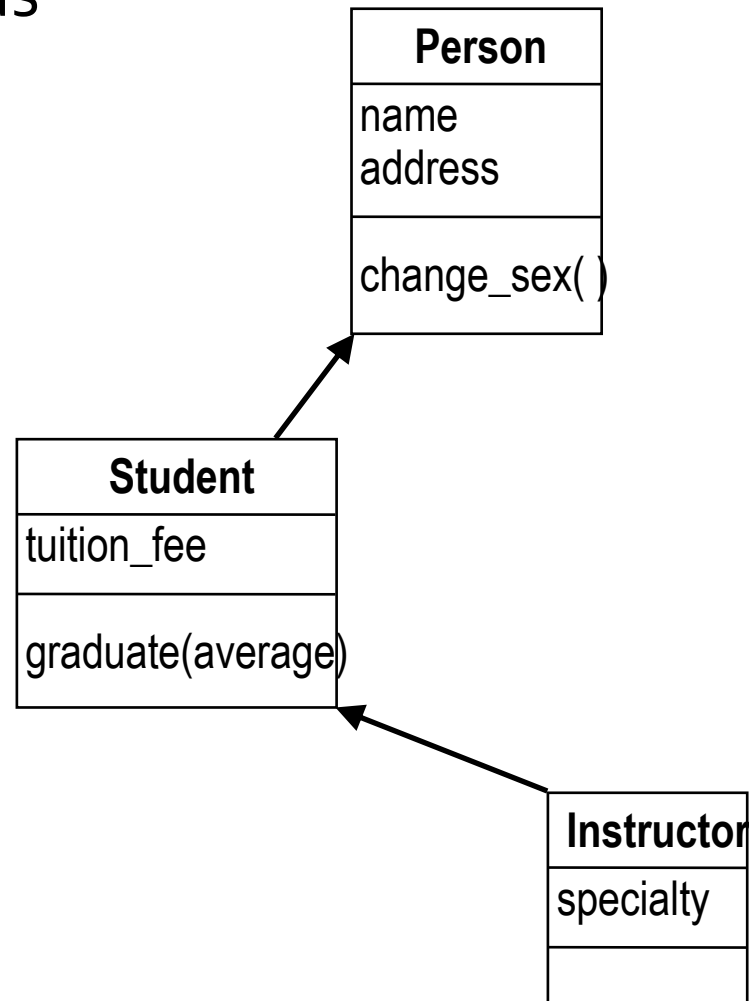
Invoking Object's clone method on an instance that does not implement the Cloneable interface results in the exception CloneNotSupportedException being thrown.

By convention, classes that implement this interface should override Object.clone (which is protected) with a public method.

See Object.clone() for details on overriding this method.

Inheritance *and* subclassing

- ▶ Inheritance can be used as a crude mechanism of reuse
- ▶ A very problematic and dangerous tactic



In this lecture we have looked at

- ▶ What is an object
- ▶ What is a class
- ▶ Advantages of OO
- ▶ OO mechanisms of
 - ▶ Encapsulation/Information Hiding
 - ▶ Inheritance
- ▶ In the class we will look at Polymorphism
 - ▶ Overloading
 - ▶ Dynamic binding
 - ▶ Genericity

Class exercise: Polymorphism

Exercises: OO inheritance

- ▶ What rules describe the relationship between a subclass and its superclass?
- ▶ For each one of the following class pairs, determine the appropriate kind of inheritance relation between them:
 - ▶ Person, Parent
 - ▶ Person, Mammal
 - ▶ Bird, FlyingObject
- ▶ What is the difference between generalization and subtyping?

Polymorphism

- ▶ **Definition:** refers to a programming language's ability to process objects differently depending on their data type or class. More specifically, it is the ability to redefine methods for derived classes.
- ▶ Polymorphism allows one message to be sent to objects of different classes
- ▶ Sending object need not know what kind of object will receive the message
- ▶ Each receiving object knows how to respond appropriately
- ▶ For example, a 'resize' operation in a graphics package
 - ▶ the way that circles, rectangles, bitmaps, groups of objects, etc will all need their own distinct methods to implement the operation. However, from the perspective of a user (or indeed a boundary object) all are called in the same way.



Polymorphism

- ▶ Polymorphic vs. monomorphic programming languages:
 - ▶ In a monomorphic language “every value, variable, and operation can be interpreted to be of one and only one type.”
 - ▶ In polymorphic languages “some values, variables, and operations may have more than one type.”
- ▶ Kinds of polymorphism:
 - ▶ Overloading
 - ▶ Dynamic binding
 - ▶ Genericity

Polymorphism I: Overloading

- ▶ Operations and methods with same name are defined for arguments of different types

```
class example {  
    void demo(int a, int b, double x, double y) {  
        a / b; // integer division  
        x / y; // floating-point division  
    }  
  
    void demo() { // method overloading  
        ...  
    }  
}
```

When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.

Thus, overloaded methods must differ in the type and/or number of their parameters.

While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.

When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Polymorphism II: Dynamic binding

- ▶ When a message is dispatched at run time, the operation that is invoked is chosen according to the class of the object. This is known as **dynamic (late) binding**.
- ▶ All OOP languages support dynamic binding.

```
public class Parent {  
    void whoami () {  
        System.out.println("I am a parent");  
    }  
}
```

```
public class Child extends Parent {  
    void whoami () {  
        System.out.println("I am a child");  
    }  
}
```

```
public class ParentTest {  
    public static void main(String args[]) {  
        Parent p = new Parent();    p.whoami();  
        Child c = new Child();      c.whoami();  
        Parent q = new Child();     q.whoami();  
    }  
}
```

In this example of Dynamic Binding we have used concept of method overriding. Child extends Parent and overrides its whoami() method and when we call whoami() method from a reference variable of type Parent, it doesn't call whoami() method from Parent class instead it calls whoami() method from Child subclass because object referenced by Parent type is a Child object. This resolution happens only at runtime because object only created during runtime and called dynamic binding in Java. Dynamic binding is slower than static binding because it occurs in runtime and spends some time to find out actual method to be called.

If we wanted to force a type then we would use static binding and not an object type for the variable.

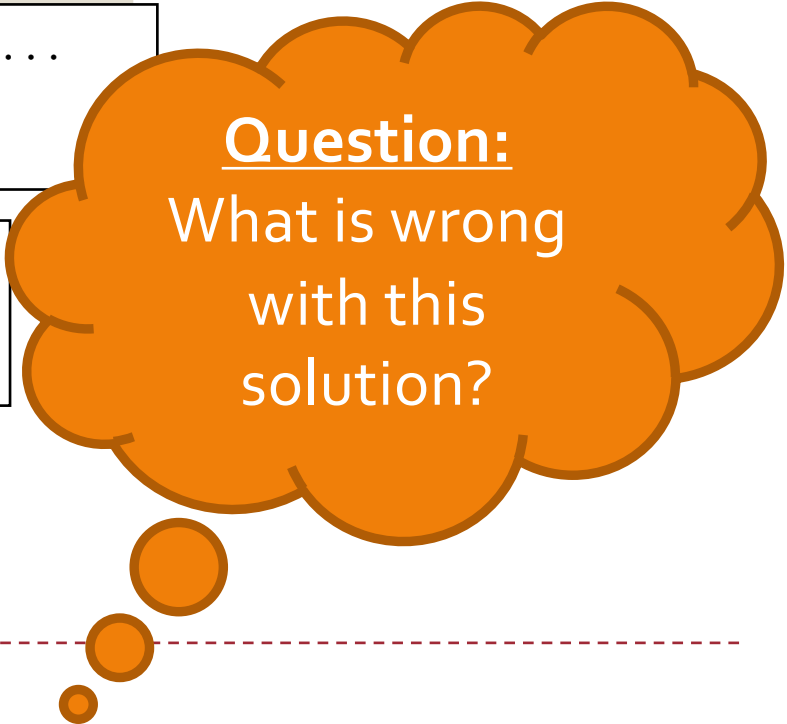
Read more: <http://javarevisited.blogspot.com/2012/03/what-is-static-and-dynamic-binding-in.html#ixzz2A7qZfS8F>

Dynamic binding: counterexample

```
public class OperatingSystem {  
    Printer myPrinter;  
    void PrintJob(Job j) {  
        switch(myPrinter.type()) {  
            case ASCII: ASCIIPrinter.print(j);  
            case PS: PSPrinter.print(j));  
            ...  
        }  
    }  
}
```

```
public class ASCIIPrinter extends Printer { ...  
    static public boolean print(Job j) {... }  
}
```

```
public class PSPrinter extends Printer { ...  
    static public boolean print(Job j) {... }  
}
```



Question:
What is wrong
with this
solution?

Dynamic binding: example

```
public class OperatingSystem {  
    Printer myPrinter;  
    void PrintJob(Job j) {  
        myPrinter.print(j)  
    }  
}
```

```
public abstract class Printer { ...  
    abstract public void boolean print(Job j);  
}
```

```
public class ASCIIPrinter extends Printer { ...  
    public boolean print(Job j) {... }  
}
```

```
public class PSPrinter extends Printer { ...  
    public boolean print(Job j) {... }  
}
```

Question:

How is this better
than the
counterexample?

Polymorphism 3: Genericity

- ▶ A mechanism allowing type-safe programming using generic classes ('templates')

```
List v = new ArrayList();  
v.add("test");  
Integer i = (Integer)v.get(0);    // Run time error
```

```
List<String> v = new ArrayList<String>();  
v.add("test");  
Integer i = v.get(0); // (type error) Compile time error
```

Example 1. The block of Java code illustrates a problem that exists when not using generics.

First, it declares an ArrayList of type Object.

Then, it adds a String to the ArrayList.

Finally, it attempts to retrieve the added String and cast it to an Integer.

Although the code compiles without error, it throws a runtime exception (java.lang.ClassCastException) when executing the third line of code.

This type of problem can be avoided by using generics and is the primary motivation for using generics.

Example 2. Using generics, the first code fragment is rewritten.

The type parameter String within the angle brackets declares the ArrayList to be constituted of String (a descendant of the ArrayList's generic Object constituents).

With generics, it is no longer necessary to cast the third line to any particular type, because the result of v.get(0) is defined as String by the code generated by the compiler.

Compiling the third line of this fragment with J2SE 5.0 (or later) will yield a compile-time error because the compiler will detect that v.get(0) returns String instead of Integer.

Exercises

- ▶ What will be the output of `ParentTest.main()`?
- ▶ In Java, is `int` a subtype of `float`? Vice versa? Is `double` a subtype of `float`? Vice versa?

Summary

- ▶ OO Analysis & Design has same mechanisms as OO programming
- ▶ OO provides many benefits
- ▶ Difference between Objects and Classes
- ▶ Looked at core OO concepts:
 - ▶ Encapsulation
 - ▶ Information Hiding
 - ▶ Inheritance
 - ▶ Generalization/specialization
 - ▶ Polymorphism
 - ▶ Overloading
 - ▶ Dynamic binding
 - ▶ Genericity

Further reading

- ▶ Object-orientation

- ▶ Chapter 4, Bennett
- ▶ Coad, P & Yourdan, E (1990) Object-oriented analysis. Prentice-Hall.
- ▶ Booch, G (1994) Object-oriented analysis and design with applications. Menlo Park.

- ▶ Information hiding & abstraction:

- ▶ Wirfs-Brock, Rebecca, Wilkerson, Brian, and Wiener, Lauren. 'Designing Object-Oriented Software'. Prentice-Hall, 1990.
- ▶ Parnas, 1985, Communications of the ACM.

- ▶ Look at Java Interfaces

- ▶ Eg. <http://docs.oracle.com/javase/tutorial/java/concepts/interface.html>
- ▶ Look at Java abstraction
 - ▶ Eg. <http://javarevisited.blogspot.co.uk/2010/10/abstraction-in-java.html>

- ▶ Exemplar based object-orientation:

- ▶ 'An exemplar based Smalltalk'. OOPSLA 1986 Proceedings.