Generalization versus Abstraction

Abstraction: simplify the description of something to those aspects that are relevant to the problem at hand.

Generalization: find and exploit the common properties in a set of abstractions.

hierarchy

polymorphism

genericity

patterns

Four-Fold Path to Generalization

Hierarchy

Exploitation of an "**is-a-kind-of**" relationship among kinds of entities to allow related kinds to share properties and implementation.

Polymorphism

Exploitation of logical or structural similarities of organization to allow related kinds to exhibit similar behaviors via similar interfaces.

Genericity

Exploitation of logical or structural similarities of organization to produce generic objects.

Patterns

Exploitation of common relationship scenarios among objects. (e.g., client/server system)

Hierarchy

Represented by generalize/specialize graph

Based on “is-a-kind-of” relationship

E.g., a Manager is an Employee; a robin is a bird, and so is an ostrich.

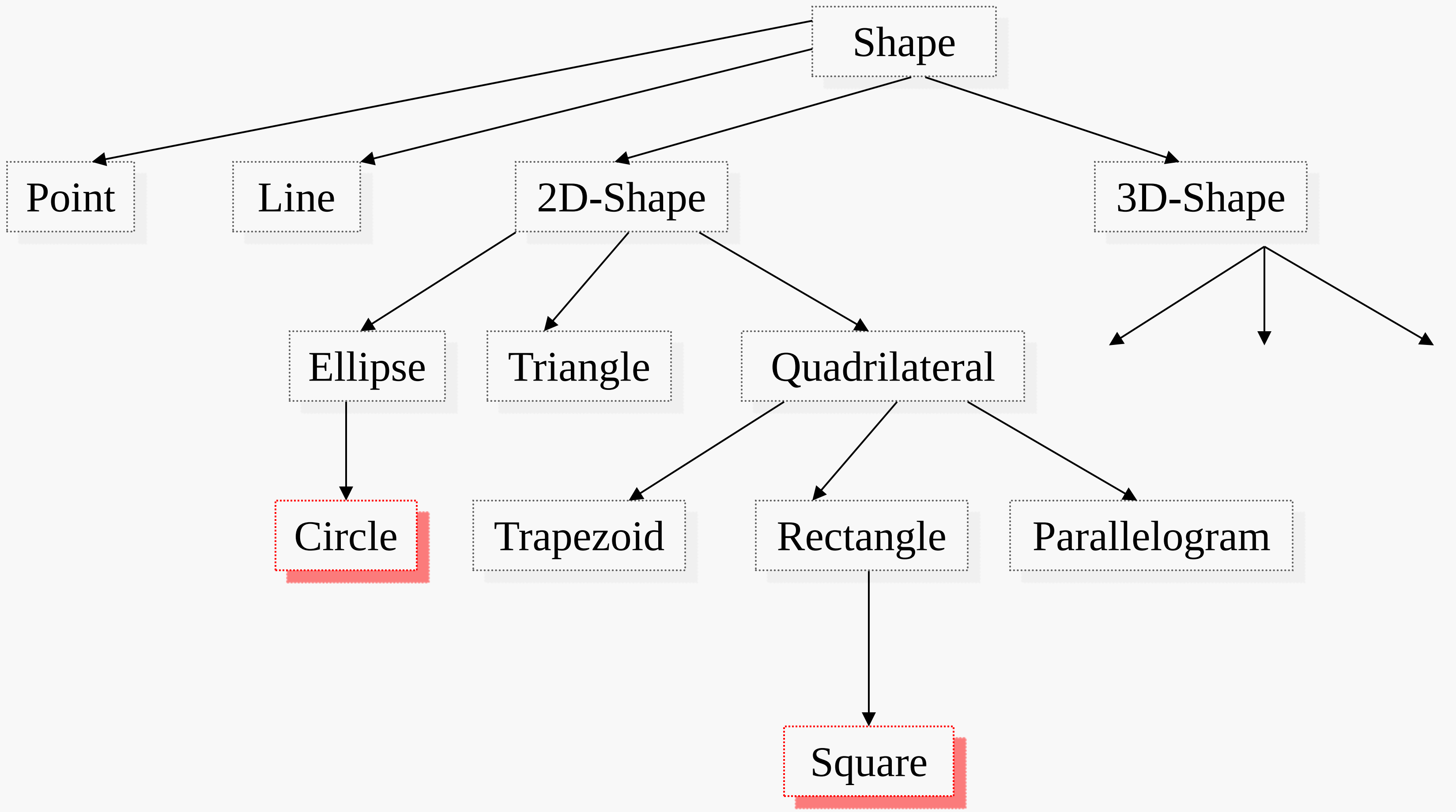
Is a form of knowledge representation – a “taxonomy” structures knowledge about nearby entities.

Extendable without redefining everything

E.g., knowing a robin is a bird tells me that a robin has certain properties and behaviors, assuming I know what a “bird” is.

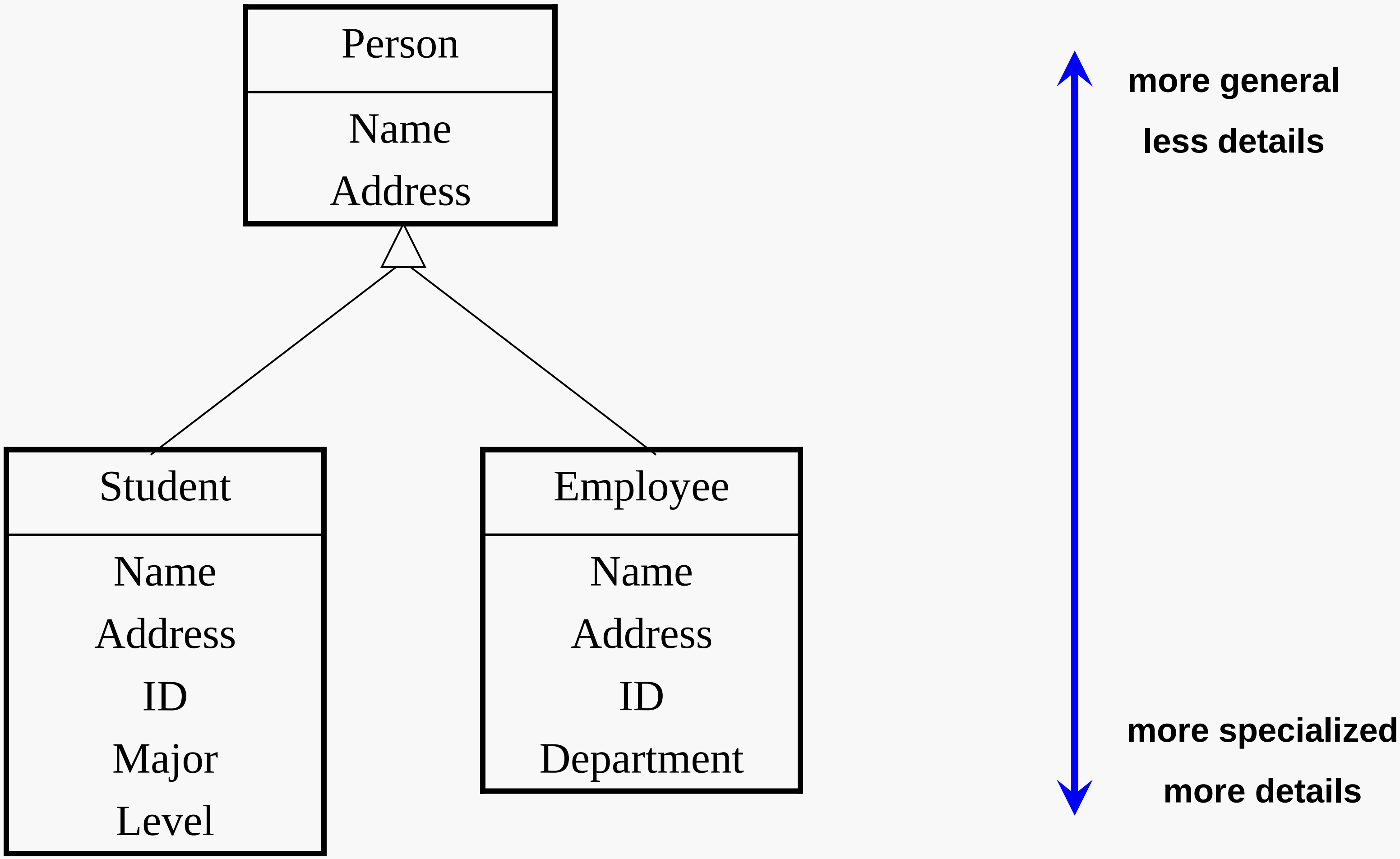
Specialization can be added to proper subset of hierarchy

A Graphics Shape Hierarchy



Taxonomy

A generalization/specialization hierarchy based on “is-a-kind-of” relationships:



Inheritance

Terminology

- Base type or class (a.k.a. superclass, parent type)
- Derived type or class (a.k.a. subclass, subtype, child type)

Important Aspects

- Programming: implement efficiently a set of related classes (mechanical)

- Design: organize coherently the concepts in an application domain (conceptual)

- Software Engineering: design for flexibility and extensibility in software systems (logical)

Student Class w/o Inheritance

```
public class Student {

  private Name    nom;
  private Address addr;
  private String  major;
  private String  ID;
  private int     level;

  public  Student(Name nom2, Address addr2, String curr,
                  String id, int rank) { ... }
  public Name     getName() { ... }
  public void     setName(Name nom2) { ... }
    //. . .
  public String   getMajor() { ... }
  public void     setMajor(String curr) { ... }
  public String   getID() { ... }
  public Student& setID(String id) { ... }
  public int      getLevel() { ... }
  public Student& setLevel(int rank) { ... }

}
```

Specify all the data members

Specify appropriate ructors

Specify accessors and mutators for all data members

`Employee` Class w/o Inheritance

```
public class Employee {

  private Name     nom;
  private Address addr;
  private String   dept;
  private String   ID;

  public Employee(Name nom2, Address addr2,
                  String office, String id) { . . . }
  public Name       getName() { . . . }
  public void       setName(Name nom2) { . . . }
    //. . .
  public String     getDept() { . . . }
  public void       setDept(String office) { . . . }
  public String     getID() { . . . }
  public void       setID(String id) { . . . }

}
```

Specify all the data members

Specify appropriate constructors

Specify accessors and mutators for all data members

What is Common?

Both classes contain the data members

```
Name    nom;
Address addr;
String  ID;
```

and the associated member functions

```
Name    getName()
Address getAddress()
String  getID()
void    setName(Name nom2)
void    setAddress (Address addr2)
void    setID(String id)
```

From a coding perspective, this is somewhat wasteful because we must duplicate the declarations and implementations in each class.

From a S/E perspective, this is undesirable since we must effectively maintain two copies of (logically) identical code.

What Do We Want?

Simply put, we want to exploit the fact that `Student` and `Employee` both are "people".

That is, each shares certain data and function members which logically belong to a more general (more basic) type which we will call a `Person`.

We would prefer to NOT duplicate implementation but rather to specify that each of the more specific types will automatically have certain features (data and functions) that are derived from (or inherited from) the general type.

Question: are there any attributes or operations in the overlap that we don't want to include in the base type `Person`?

How Do We Get It?

By employing the inheritance mechanism…

Inheritance in is NOT simple, either syntactically or semantically. We will examine a simple case first (based on the previous discussion) and defer explicit coverage of many specifics until later.

Inheritance in involves specifying in the declaration of one class that it is derived from (or inherits from) another class.

Some languages incorporate inheritance differently. The mechanics of specifying inheritance differ along with subtle forms of inheritance.

The Base Class: `Person`

Having identified the common elements shared by both classes (`Employee` and `Student`), we specify a suitable base class:

```
public class Person {

 private Name    nom;
 private Address addr;

 public Person(Name nom2, Address addr2)
        { ... }
 public Name    getName() { ... }
 public void    setName(Name& nom) { ... }
 public void    setAddress(Address addr2) { ... }
 public Address getAddress() { ... }

}
```

The base class should contain data members and function members that are general to all the types we will derive from the base class.

A Derived Class: `Employee`

Specify public inheritance

Specify base class

```
public class Employee extends Person {

 private String dept;
 private String ID;

 public Employee()
 public Employee(Person per, String office,
                 String id)
 public Employee(Name nom2, Address addr2,
                 String office, String id)

 public String    getDept()
 public void      setDept(String office)
 public String    getID()
 public void      setID(String id)

}
```

Specify additional data members not present in base class

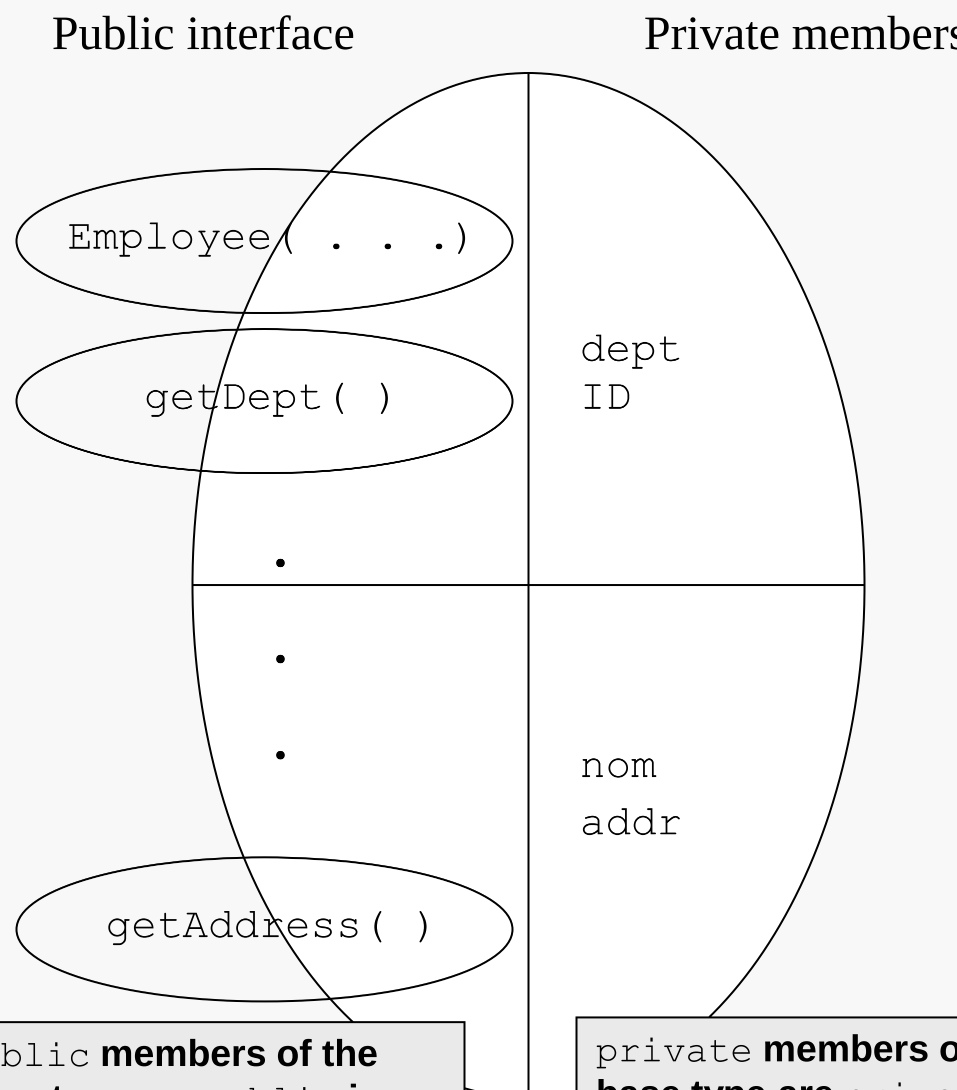
Specify appropriate constructors

Specify accessors and mutators only for the added data members

Logical View of an `Employee` Object

`Employee` "layer"

`Person` "layer" (inherited from base type)



`public` members of the base type are `public` in the derived type

`private` members of the base type are `private` in the derived type

Constructing a Derived Type Object

When an object of a derived type is declared, the default constructor for the base type will be invoked BEFORE the body of the constructor for the derived type is executed (unless an alternative action is specified…).

```
public Employee() {
   super();
   dept = "None";
   ID   = "None";
}
```

It's not necessary to explicitly invoke the base constructor here, but it makes the behavior more obvious.

Alternatively, the derived type constructor may explicitly invoke a non-default base type constructor :

```
public Employee(Person per, String office,
                    String& id) {
   super(per.getName(), per.getAddress() );
   dept = office;
   ID   = id;
}
```

Here, the (default) copy ructor for the base class is used.

Derived Class Member Access Problem

Objects of a derived type inherit the data members and function members of the base type. However, the derived object may not directly access the private members of the base type:

```
public Employee(Person per, String office,
                            String id) {

    nom  = per.getName();
    addr = per.getAddress();
    dept = office;
    ID   = id;
}
```

Error: cannot access private member declared in class `Person`.

For a derived-class constructor we directly invoke a base class constructor, as shown on the previous slide, or use the `Person` interface:

```
public Employee(Person per, String office,
                            String id) {
    setName(per.getName());
    setAddress(per.getAddress());
    // . . .
}
```

Protected Access

The restriction on a derived type's access seems to pose a dilemma:

- Having the base type use only public members is certainly unacceptable.
- Having the derived class use the public interface of the base class to access and/or modify private base class data members is clumsy.

Java/C++ provides a middle-ground level of access control that allows derived types to access base members which are still restricted from access by unrelated types.

The keyword `protected` may be used to specify the access restrictions for a class member:

```
public class Person {

 protected Name    nom;
 protected Address addr;
//. . .
}
```

→

```
public Employee( /*. . .*/ )
{

   nom    = nom2;      // OK now
   addr   = addr2;
   dept   = office;
   ID     = id;
}
```

A Sibling Class

```
public class Student extends Person {

 private String major;
 private String ID;
 private int    level;


 public Student(Person per,
           String curr,
           String id, int rank)

 public String   getMajor()
 public void     setMajor(String curr)
 public String   getID()
 public void     setID(String id)
 public int      getLevel()
 public void     setLevel(int rank)

}
```

Note that, so far as the language is concerned, `Student` and `Employee` enjoy no special relationship as a result of sharing the same base class.

Using Objects of Derived Classes

Objects of a derived class may be declared and used in the usual way:

```
//. . .
Person JBH = new Person(new Name("Joe", "Bob", "Hokie"),
             new Address("Oak Bridge Apts", "#13", "Blacksburg",
                   "Virginia", "24060"));

Employee JoeBob = new Employee(JBH, "Sales", "jbhokie");


System.out.println("Name:   " + JoeBob.getName() +
      "  Dept:   " + JoeBob.getDept()
      "  ID:     " + JoeBob.getID() );

Person HHooI = new Person(new Name("Haskell", "Horatio", "Hoo"),
               new Address("1 Rotunda Circle", "",
                   "Charlottesville", "VA", "21009"));
Student HaskellHoo = new Student(HHooIV, "Undecided",
                                 "101-01-0101", 40);

HaskellHoo.setAddress(new Address("Deke House", "333 Coors Way",
                               "Charlottesville", "VA",
                               "21010"));
HaskellHoo.setMajor("Undeclared");
//. . .
```

Call base member

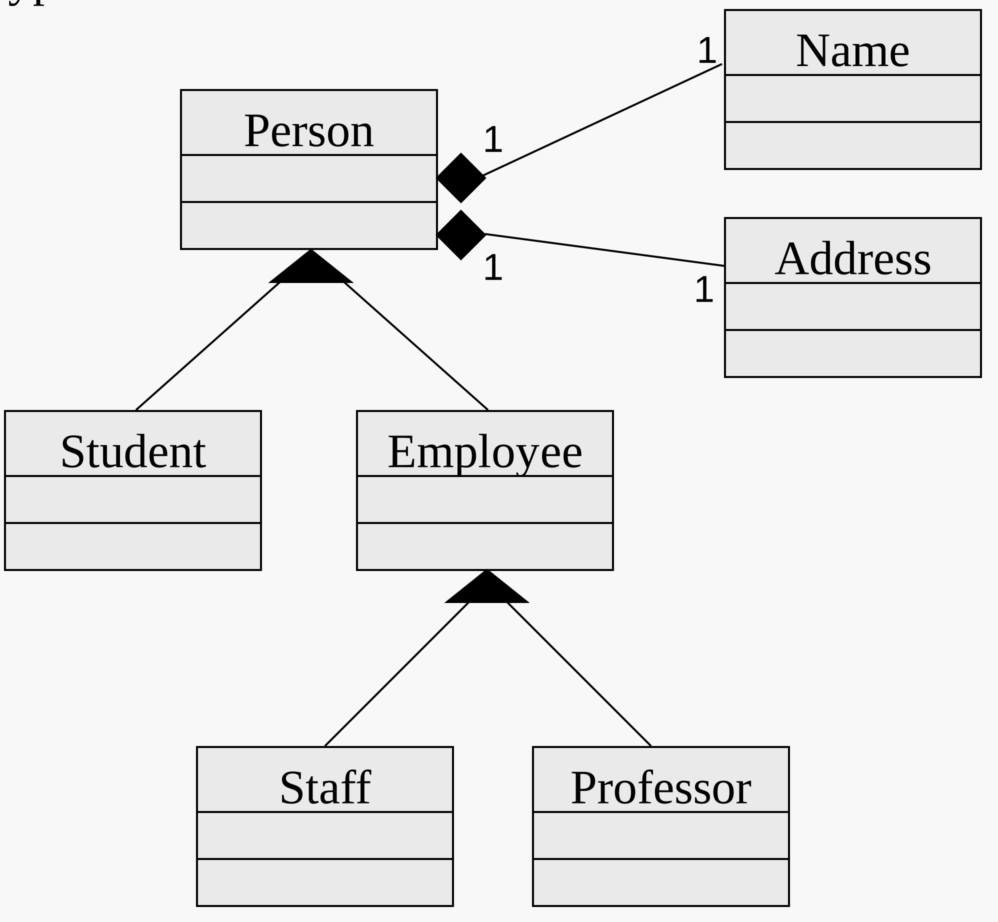
Call derived members

Base object is only declared to simplify constructor call.

…to the client code there's no evidence here that the class is derived…

Extending the Hierarchy

Actually, `Employee` is not a terribly interesting class but it has two (or more) useful sub-types:



There's no restriction on how many levels of inheritance can be designed, nor is there any reason we can't mix inheritance with association and/or aggregation.

Staff Class and Professor Class

For the sake of an example, a staff member is paid an hourly wage, so the class `Staff` must provide the appropriate extensions…

```
public class Staff extends Employee {

 private double hourlyRate;

 public Staff(Employee emp, double rate)
 public double getRate()
 public void   setRate(double rate)
 public double grossPay(int hours)

}
```

…whereas a professor is paid a fixed salary:

```
public class Professor extends Employee {

 private double salary;

 public Professor(Employee emp, double income)
 public double getSalary()
 public void   setSalary(double income)
 public double grossPay(int days)

}
```


An Inadequate Base Class Member Function

The base member function `Employee setID()` is simple:

```
public void setID(String id)
{
    ID = id;
    // return (this);   //chaining
}
```

This implementation raises two issues we should consider:

- What if there's a specialized way to set the ID field for a derived type?
- Is the return type really acceptable for a derived type?

We'll consider the first question now… suppose that the ID for a professor must begin with the first character of that person's department.

Then `Professor setID()` must enforce that restriction.

Overriding a Base Class Member Function

In the derived class, provide an appropriate implementation, using the same interface. That will override the base class version when invoked on an object of the derived type:

```
@Override
public void setID(String id) {

   if ( id.charAt(0) == dept.charAt(0) )
      ID = id;
   else
      ID = dept.charAt(0) + id;

}
```

Assuming that dept has protected status.

The appropriate member function implementation is chosen (at compile time), based upon the type of the invoking object and the inheritance hierarchy. Beginning with the derived class, the hierarchy is searched upward until a matching function definition is found:

```
Employee  E = new Employee( /*. . .*/ );
Professor F = new Professor(/*. . .*/ );
//. . .
E.setID("12334");   // Employee setID()
F.setID("99012");   // Professor setID()
```

Extending a Base Class Member Function

Suppose we added a display member function to the base type:

```
void display(PrintWriter out) {

   out.print("Name:  " + nom +
        "  Address: " + addr);
}
```

This is inadequate for a `Professor` object since it doesn't recognize the additional data members… we can fix that by overriding again (with a twist):

```
void display(PrintWriter out) {

   super.display(Out);

   out.print("ID:     " + ID +
        "  Dept: " + dept);
};
```

Here, we use the base class display function, invoking it with the appropriate scope resolution, and then <u>extend</u> that implementation with the necessary additional code.

Copying a Derived Object to a Base Object

It is legal to assign a derived type object to a base type object:

```
Employee eHomer = new Employee(new Name("Homer", "P", "Simpson"),
 new Address("1 Chernenko Way", "", "Blacksburg", "VA", "24060"),
             "Physics", "P401" );
Professor homer = new Professor(eHomer, 45000.00);

Employee emp;
Person   per;

emp = homer;  // legal assignments, but usually inadvisable
per = homer;
```

```
System.out.println( per.getName() );

System.out.println( emp.getDept() );

//invalid
System.out.println( per.getSalary() );

System.out.println( emp.grossPay(14) );
```

When a derived object is assigned to a base target, only the public members appropriate to the target type are accessible.

Another Slice

```
void printEmployee(Employee toPrint, PrintWriter out)
{
   out.print( toPrint.getID() + "\" + toPrint.getName() );
}
```

`printEmployee()` sees only the `Employee` layer of the actual parameter that was passed to it.

That's actually OK in this case since that's all `printEmployee()` deals with anyway.

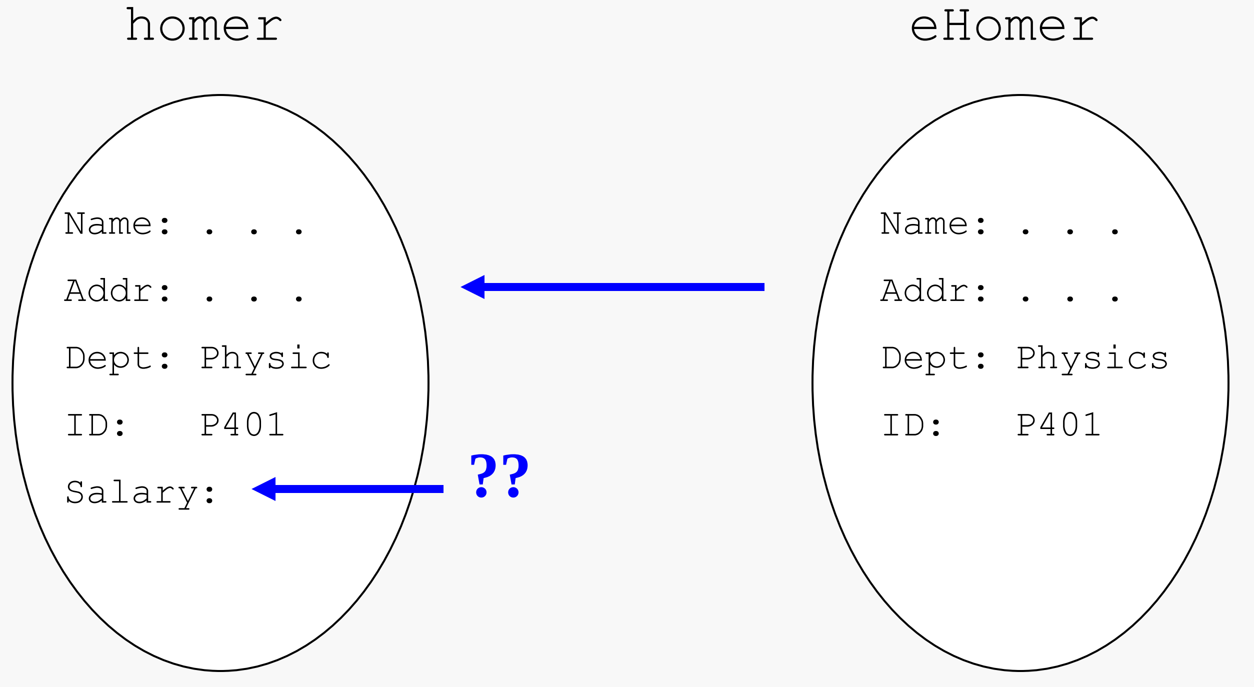
However, it's certainly a limitation you must be aware of… what if you wanted to write a generic print function that would accept any derived type?

Assigning Base Type to Derived Type

By default, a base type object may **not** be assigned to a derived type object:

```
// assume declarations from slide 25. . .

homer = eHomer;     // illegal - compile time error
```



Inheritance and Development

Inheritance provides a number of benefits with respect to development:

- reusability of common implementation
- representation of natural logical relationships among types

Inheritance also carries a cost:

- designing modifications to base class require understanding the effect on <u>all</u> derived classes
- designing modifications to derived class requires understanding of the relationship to the base class (not usually too serious)
- modifications to base class will require re-testing implementations of derived classes to verify nothing is broken