OOP and Inheritance

Programming Languages

CS 214

Introduction

Let's play the children's game: *Twenty Questions*

→ You have 20 questions to guess what I'm thinking about...

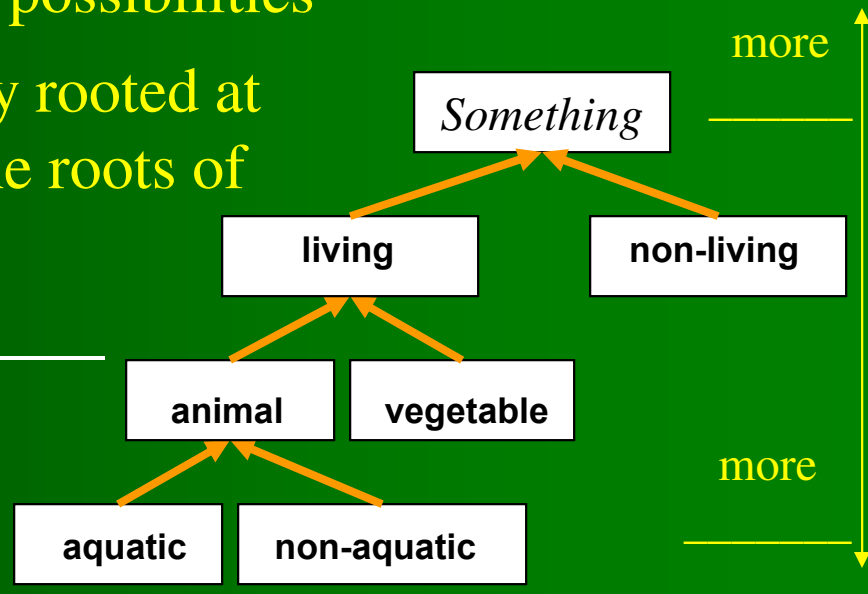
What are you trying to do with your questions?

→ *Good* questions eliminate large classes of objects to narrow the number of remaining possibilities

→ The game presupposes a hierarchy rooted at *Something*, whose subclasses are the roots of less general class hierarchies:

→ A good question lets you move down in the hierarchy...

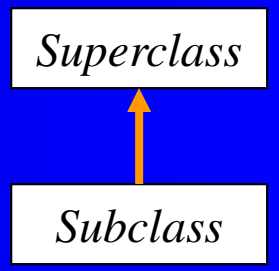
We seem to pick this hierarchy up quite early (as children)...



Object-Oriented Programming (OOP)

One of the basic aims of the class is to allow programmers to model objects (abstract or concrete) from the real world.

→ OOP supports *hierarchical class relationships*:



→ Each ↑ represents the is-a relationship, indicating that the *subclass* inherits the attributes of its *superclass*.

→ *Object-oriented analysis & design* (OOD) uses superclasses and inheritance to consolidate attributes that different objects have in common, so that those attributes need not be defined more than once.

Different OO languages have different conventions for representing inheritance, but the concept is the same.

Example: A Payroll Problem

Suppose we have these kinds of workers on our payroll:

- Faculty member
  - name
  - id number
  - dept
  - salary
  - research specialty
- Staff member
  - name
  - id number
  - dept
  - hourly rate
  - hours worked
  - supervisor
- Administrator
  - name
  - id number
  - dept
  - salary
- Student worker
  - name
  - id number
  - dept
  - hourly rate
  - hours worked

How can we organize these so as to avoid redundant code?

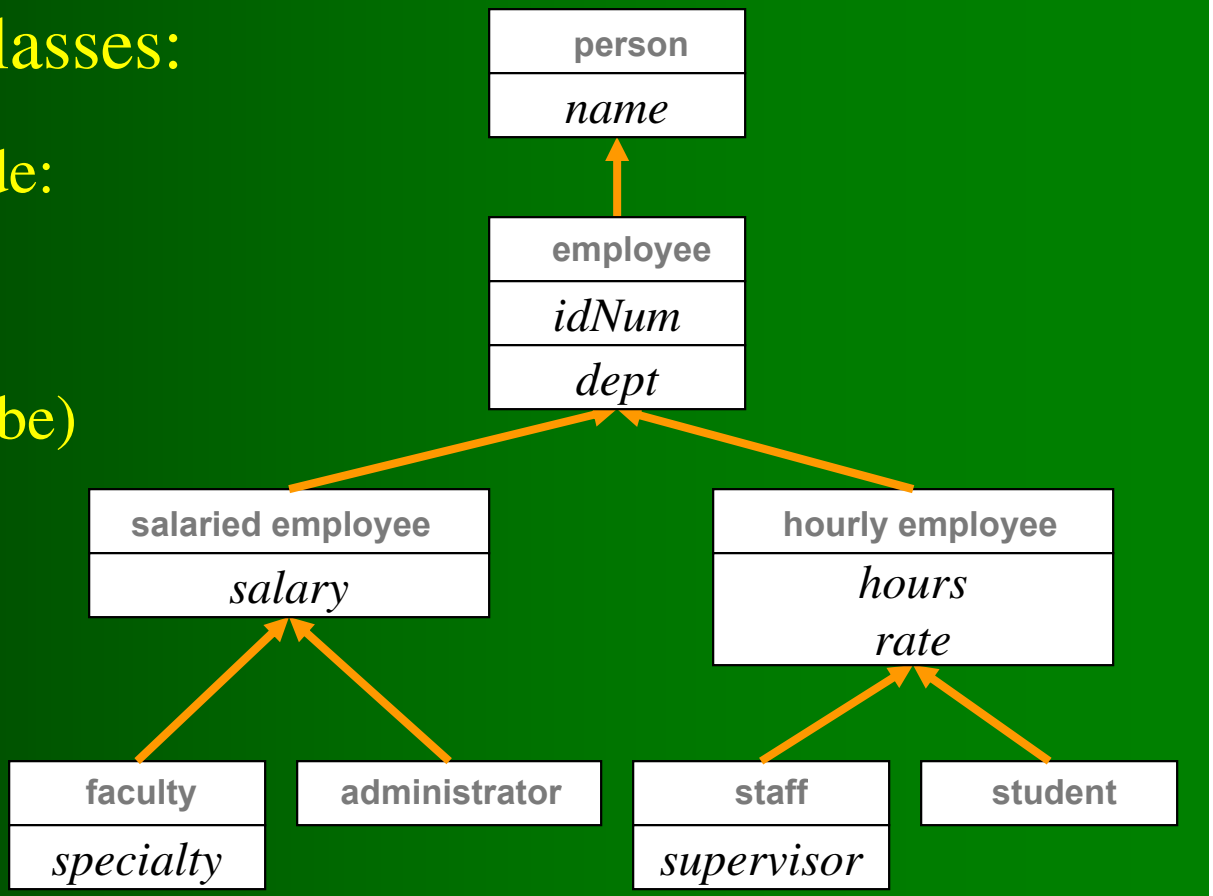
Design

We can start with the 'leaf' classes and consolidate common attributes into superclasses:

Each class should provide:

- *Constructors*
- *Accessor* methods
- *Mutator* methods (maybe)
- *I/O* methods
- *pay* method (*Employee* and below)

Note that our design process is bottom-up, not top-down...



Implementation: C++

Given a design, our ____implementation____ proceeds ____top-down____:

```
class Person {
 public:
   Person();
   Person(string name);
   string getName() const;
   virtual void write(ostream& out) const;
   virtual void read(istream& in);
   friend ostream& operator<<(ostream & out, const Person & p);
   friend istream& operator>>(istream & in, Person & p);

 private:
   string myName;
};
```

In order for subclasses to override read() and write() with their own definitions, these must be declared as ____virtual____ methods in C++.

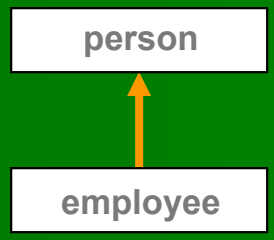
Implementation: C++ (ii)

```
inline Person::Person() { myName = ""; }

inline Person::Person(string name) { myName = name; }

inline string Person::getName() const { return myName; }

inline void Person::write(ostream& out) const
 { out << myName << endl; }

inline void Person::read(istream& in) { getline(in, myName); }

inline ostream& operator<<(ostream & out, const Person& p)
  { p.write(out); return out; }

inline istream& operator>>(istream& in, Person& p)
  { p.read(in); return in; }
```

Each of these is simple enough to define *inline* in C++ (i.e., in the header file)...

Implementation: C++ (iii)

We continue with the *Employee* subclass of *Person*:




```
class Employee : public Person {
 public:
   Employee();
   Employee(string name, int id, string dept);
   int getID() const;
   string getDept() const;
   virtual void write(ostream& out) const;
   virtual void read(istream& in);
   virtual double pay() const = 0;
 private:
   int    myID;
   string myDept;
};
```

pay() is a pure vitual function because every *Employee* should respond to that message, but its subclasses must supply its definition.

Implementation: C++ (iv)

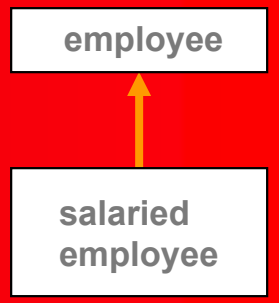
```
inline Employee::Employee()
  : Person()
{ myID = 0; myDept = ""; }

inline Employee::Employee(string name, int id, string dept)
  : Person(name)
{ myID = id; myDept = dept; assert(id > 0); }

inline int Employee::getID() const { return myID; }

inline string Employee::getDept() const { return myDept; }

inline void Employee::write(ostream& out) const {
   Person::write(out);
   out << myID << endl << myDept << endl;
}

inline void Employee::read(istream& in) {
   Person::read(in);
   in >> myID >> myDept; assert(myID > 0);
}
```

Employee inherits << and >>, and since they call *write()* and *read()*, we need not redefine them...

Implementation: C++ (v)

We continue with *Employee*'s *SalariedEmployee* subclass:

```
class SalariedEmployee : public Employee {
 public:
   SalariedEmployee();
   SalariedEmployee(string name, int id,
                    string dept, double salary);
   double getSalary() const;
   virtual void write(ostream& out) const;
   virtual void read(istream& in);
   virtual double pay() const;

 private:
   double    mySalary;
};
```



A *SalariedEmployee* has the information needed to compute its pay, so it supplies the definition for pay().

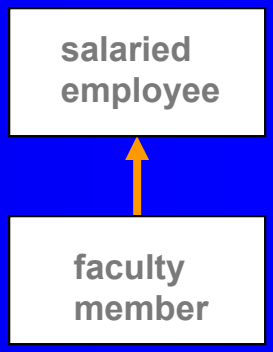
Implementation: C++ (vi)

```
inline SalariedEmployee::SalariedEmployee()
  : Employee()
{ mySalary = 0.0; }

inline SalariedEmployee::SalariedEmployee(string name, int id,
                                          string dept, double salary)
  : Employee(name, id, dept)
{ mySalary = salary; assert(mySalary > 0.0); }

inline double SalariedEmployee::getSalary() const
{ return mySalary; }

inline void SalariedEmployee::write(ostream& out) const
{ Employee::write(out);
  out << mySalary << endl; }

inline void SalariedEmployee::read(istream& in)
{ Employee::read(in);
  in >> mySalary; assert(mySalary > 0.0); }

inline double SalariedEmployee::pay() const
{ return mySalary;}
```


Implementation: C++ (vii)

We continue with the *FacultyMember* subclass:

```
class FacultyMember : public SalariedEmployee {
 public:
  FacultyMember();
  FacultyMember(string name, int id, string dept,
                double salary, string specialty);
  string getSpecialty() const;
  virtual void write(ostream& out) const;
  virtual void read(istream& in);

 private:
  string    mySpecialty;
};
```



A *FacultyMember* inherits the *name-*, *id-*, *department-*, and *salary*-related attributes/methods from its superclass.

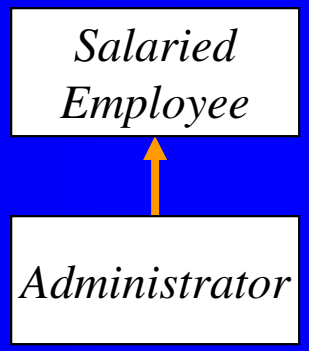
Implementation: C++ (viii)

```
inline FacultyMember::FacultyMember()
  : SalariedEmployee()
{ mySpecialty = ""; }

inline FacultyMember::FacultyMember(string name, int id,
                                    string dept, double salary,
                                    string specialty)
  : SalariedEmployee(name, id, dept, salary)
{ mySpecialty = specialty; }

inline string FacultyMember::getSpecialty() const
{ return mySpecialty; }

inline void FacultyMember::write(ostream& out) const
{ SalariedEmployee::write(out);
  out << mySpecialty << endl; }

inline void FacultyMember::read(istream& in)
{ SalariedEmployee::read(in);
  in >> mySpecialty; }
```

We then do the same for the other classes in our design.

Implementation: C++ (ix)

The *Administrator* class is especially easy:




```
class Administrator : public SalariedEmployee {
 public:
   Administrator();
   Administrator(string name, int id, string dept,
                 double salary);
};

inline Administrator::Administrator() : SalariedEmployee()
{}

inline Administrator::Administrator(string name, int id,
                                    string dept, double salary)
  : SalariedEmployee(name, id, dept, salary)
{}
```

Our *Administrator* class is this simple because it has no attributes/methods beyond those it inherits from its superclass…

Use: C++

Given our hierarchy, we can write something like this:

```
// …
ifstream fin("payroll.data");
Employee* empPtr; char empType;
for (;;) {
   fin >> empType;
   if ( fin.eof() ) break;
   switch (empType) {
    case 'A': empPtr = new Administrator(); break;
    case 'F': empPtr = new FacultyMember(); break;
    case 'S': empPtr = new StaffMember(); break;
    case 'W': empPtr = new StudentWorker(); break;
   }
   fin >> (*empPtr);  // equivalent to empPtr->read(fin);
   cout << empPtr->getName() << endl
        << empPtr->pay() << endl;
}
fin.close();
// …
```

Our variable *empPtr* is called a handle, because it can 'grab' different objects...

Compile-Time vs Run-Time Binding

In C++, the *virtual* keyword tells the compiler to _____ to bind messages to their definition (by default, binding occurs at _____ in C++).

If we don't declare prototypes of *write()* as *virtual*:

```
class Employee {
 public:
   // …
   void write(ostream& out) const;
   // …
};
```

then subsequent calls to *write()*:

```
Employee* empPtr;
// …
empPtr->write(cout);
```

are statically bound to *Employee::write()* at _____ (because the handle is an *Employee**) instead of being dynamically bound to the receiver's *write()* at _____.

Polymorphism

By declaring *read()* and *write()* as *virtual*:

```
class Person {
  // …
  virtual void write(ostream& out) const;
  virtual void read(istream& in);
  // …
};
```

subsequent calls to these methods:

```
Employee* empPtr;
// …
empPtr->write(cout);
```

are bound to the *receiver's* definitions of those methods at *run-time*.

- The same call to *write()* may thus invoke *FacultyMember::write()*, *Administrator::write()*, *StaffMember::write()* or *StudentWorker::write()* depending on the object to which the handle *empPtr* points.
- This behavior is called polymorphic behavior, or polymorphism.
- dynamic dispatch (aka *runtime binding*) is the mechanism by which a message is bound according to *the receiver's type*, instead of the handle's type.

Implementation: Java

Let's compare our C++ implementation to Java:

```
public class Person {
 public Person() { myName = ""; }
 public Person(String name) { myName = name; }
 public final String getName() { return myName; }
 public void write(PrintWriter out) { out.println(myName); }
 public void read(BufferedReader in) { myName = in.readLine(); }

 private String myName;
}
```

Java has no operator overloading, no const methods and no friends.

In C++, compile-time binding is the default; run-time binding (polymorphism) must be enabled using the *virtual* keyword.

In Java, _____; compile-time binding must be enabled using the _____ keyword.

Implementation: Java (ii)

Continuing with the *Employee* subclass of *Person*:

```
abstract class Employee extends Person {
  public Employee() { super(); myID = 0; myDept = ""; }

  public Employee(String name, int id, String dept)
  { super(name); myID = id; myDept = dept; }

  public final int getID() { return myID; }

  public final String getDept() { return myDept; }

  public void write(PrintWriter out)
  { super.write(out); out.println(myID); out.println(myDept); }

  public void read(BufferedReader in)
  { super.read(in); String idString = in.readLine();
    myID = Integer.parseInt(idString); myDept = in.readLine(); }

  abstract public double pay();     // "pure virtual" in Java
  private int    myID;
  private String myDept;
}
```


Implementation: Java (iii)

```
class SalariedEmployee extends Employee {
 public SalariedEmployee() { super(); mySalary = 0.0; }

 public SalariedEmployee(String name, int id,
                         String dept, double salary)
 { super(name, id, dept); mySalary = salary; }

 public final double getSalary() { return mySalary; }

 public void write(PrintWriter out)
  { super.write(out); out.println(mySalary); }

 public void read(BufferedReader in)
  { super.read(in); String salaryString = in.readLine();
    mySalary = Double.parseDouble(salaryString); }

 public double pay() { return mySalary; }

 private double mySalary;
}
```

Java lets us do most of the same things, but (usually) more easily...

Implementation: Java (iv)

```
class FacultyMember extends SalariedEmployee {

 public FacultyMember() { super(); mySpecialty = ""; }

 public FacultyMember(String name, int id, String dept,
                      double salary, String specialty)
  { super(name, id, dept, salary); mySpeciality = specialty; }

 public final String getSpecialty() { return mySpecialty; }

 public void write(PrintWriter out)
  { super.write(out); out.println(mySpecialty); }

 public void read(BufferedReader in)
  { super.read(in); mySpecialty = in.readLine(); }

 private String mySpecialty;
}
```

We then implement the other classes the same way...

Implementation: Java (v)

As before, *Administrator* indicates how easy this is:

```
class Administrator extends SalariedEmployee {

 public Administrator() { super(); }

 public Administrator(String name, int id, String dept,
                      double salary)
   { super(name, id, dept, salary); }

}
```

Our *Administrator* class is this simple because it has no attributes/methods beyond those it inherits from its superclass…

Use: Java

To use these classes, we can write something like this:

```
// …
BufferedReader fin = new BufferedReader(
                       new InputStreamReader(
                         new FileReader("payroll.data")));
Employee emp = null; String eType = null;
for (;;) {
  eType = fin.readLine();          // name of class
  if ( eType == null ) break;
  Employee emp = (Employee) Class.forName(eType).newInstance();
  emp.read(fin);
  System.out.println( emp.getName() + "\n" + emp.pay() );
}
fin.close();
// …
```

All non-primitive-type variables are _____ in Java.

Java's *Class* class provides a very convenient way to build an instance of a class from a string whose value is the name of the class.

Implementation: Ada

Let's compare Ada to our other implementations:

```
package PersonPackage is
   type Person is tagged private;
   type PersonRef is access all Person'Class;
   procedure Init(P: in out Person; AName: Unbounded_String);
   function GetName(P: in Person) return Unbounded_String;
   procedure Read(F: in out File_Type; P: in out Person);
   procedure Write(F: in out File_Type; P: in Person);
   procedure Put(F: in out File_Type; P: in Person'Class);
   procedure Get(F: in out File_Type; P: in out Person'Class);
private
   type Person is tagged record
        itsName : Unbounded_String;
    end record;
end PersonPackage;
```

In Ada, a *subtype* can inherit from a *tagged* type (for polymorphism);

and a *handle* is declared as a pointer to a *Class-wide* type.

Implementation: Ada (ii)

Our package body is as follows:

```
package body PersonPackage is
  procedure Init(P: in out Person; AName: Unbounded_String) is
   begin
      P.ItsName := AName;
  end Init;
  function GetName(P: in Person) return Unbounded_String is
   begin
      return P.ItsName;
  end GetName;
  procedure Write(F: in out File_Type; P: in Person) is begin
      Put(F, P.ItsName); New_Line(F);
  end Write;
  procedure Put(F: in out File_Type; P: in Person'Class) is
   begin
         Write(F, P);    -- P is class-wide -> dynamic dispatch
  end Put;
  -- … Read, Get are similar …
end PersonPackage;
```


Implementation: Ada (iii)

We then build *Employee* as an extension of *Person*:

```
package EmployeePackage is
  type Employee is abstract new Person with private;
  type EmployeeRef is access all Employee'Class;
  procedure Init(E: in out Employee; name: Unbounded_String;
                 id: Integer; dept: Unbounded_String);
  function GetID(E: in Employee) return Integer;
  function GetDept(E: in Employee) return Unbounded_String;
  procedure Write(F: in out File_Type; E: in Employee);
  procedure Read(F: in out File_Type; E: in out Employee);
  function GetPay(E: in Employee'Class) return float;
  function Pay(E: in Employee) return float is abstract;

private
   type Employee is abstract new Person with record
       itsID : Integer;
       itsDept : Unbounded_String;
    end record;
end EmployeePackage;
```


Implementation: Ada (iv)

```
package body EmployeePackage is
 procedure Init(E: in out Employee; Name: in Unbounded_String;
                Id: in Integer; Dept: in Unbounded_String)
  is begin
    Init(Person(E), Name); E.ItsID := Id; E.ItsDept := Dept;
 end Init;
 function GetId(E: in Employee) return Integer is begin
    return Emp.ItsId;
 end GetId;
 -- … GetDept() is similar …
 procedure Write(F: in out File_Type; E: in Employee) is begin
    Write(F, Person(E));
    Put(F, E.ItsId); New_line(F);
    Put(F, E.ItsDept); New_Line(F);
 end Write;
 -- … read(F, E) is similar; Get(F,E), Put(F,E) are not needed!
 function GetPay(E: in Employee'Class) return float is begin
    return Pay(E);     // E is class-wide -> dynamic dispatch
 end GetPay;
end EmployeePackage;
```


Implementation: Ada (v)

We then build *SalariedEmployee* as an extension of *Employee*:

```
package SalariedEmployeePackage is
 type SalariedEmployee is new Employee with private;
 type SalariedEmployeeRef is access all SalariedEmployee'Class;

 procedure Init(sE: in out SalariedEmployee;
                Name: in Unbounded_String; Id: in Integer;
                Dept: in Unbounded_String; Salary: in Float);

 function GetSalary(sE: in SalariedEmployee) return Float;

 procedure Write(F: in out File_Type; sE: in SalariedEmployee);

 procedure Read(F: in out File_Type;
                sE: in out SalariedEmployee);

 function Pay(sE: in SalariedEmployee) return Float;

private
  type SalariedEmployee is new Employee with record
      itsSalary : Float;
   end record;
end SalariedEmployeePackage;
```


Implementation: Ada (vi)

```
package body SalariedEmployeePackage is
 procedure Init(sE: in out SalariedEmployee;
                Name: in Unbounded_String; Id: in Integer;
                Dept: in Unbounded_String; Salary: in Float)
  is begin
    Init(Employee(sE), Name, Id, Dept); sE.ItsSalary := Salary;
 end Init;
 function GetSalary(sE: in SalariedEmployee) return Float is
  begin
    return sE.ItsSalary;
 end GetSalary;
 procedure Write(F: in out File_Type; sE: out SalariedEmployee)
  is begin
    Write(F, Employee(sE));
    Put(F, sE.ItsSalary); New_line(F);
 end Write;
 -- … Read(F, sE) is similar…
 function Pay(sE: in SalariedEmployee) return Float is begin
    return mySalary;
 end Pay;
end SalariedEmployeePackage ;
```


Implementation: Ada (vii)

We then build *Faculty* as an extension of *SalariedEmployee*:

```
package FacultyPackage is
 type Faculty is new SalariedEmployee with private;
 type FacultyRef is access all Faculty'Class;

 procedure Init(F: in out Faculty; Name: in Unbounded_String;
                Id: in Integer; Dept: in Unbounded_String;
                Salary: in Float; Specialty: in Unbounded_String);

 function GetSpecialty(F: in Faculty) return Unbounded_String;

 procedure Write(outf: in out File_Type; F: in Faculty);

 procedure Read(inF: in out File_Type; F: in out Faculty);

private
  type Faculty is new SalariedEmployee with record
      itsSpecialty : Unbounded_String;
   end record;
end FacultyPackage;
```


Implementation: Ada (viii)

```
package body FacultyPackage is
 procedure Init(F: in out Faculty; Name: in Unbounded_String;
                Id: in Integer; Dept: in Unbounded_String;
                Salary: in Float; Specialty: in Unbounded_String)
  is begin
    Init(SalariedEmployee(F), Name, Id, Dept, Salary);
    F.ItsSpecialty := Specialty;
 end Init;

 function GetSpecialty(F: in Faculty) return Unbounded_String
  is begin
    return F.ItsSpecialty;
  end GetSpecialty;

 procedure Write(outf: in out File_Type; F: in Faculty) is
  begin
    Write(outf, SalariedEmployee(F));
    Put(outf, F.ItsSpecialty); New_Line(F);
  end Write;

  -- … Read(outf, F) is similar…
end FacultyPackage ;
```


Implementation: Ada (ix)

We then build *Administrator* as an extension of *SalariedEmployee*:

```
package AdministratorPackage is
 type Administrator is new SalariedEmployee with private;
 type AdministratorRef is access all Administrator'Class;

private
  type Administrator is new SalariedEmployee with record
      null;
   end record;
end AdministratorPackage;
```

Class *Administrator* inherits everything it needs from its superclass *SalariedEmployee*, so its package body is empty:

```
package body AdministratorPackage is
   -- empty body; Administrator defines no new attributes
end AdministratorPackage ;
```

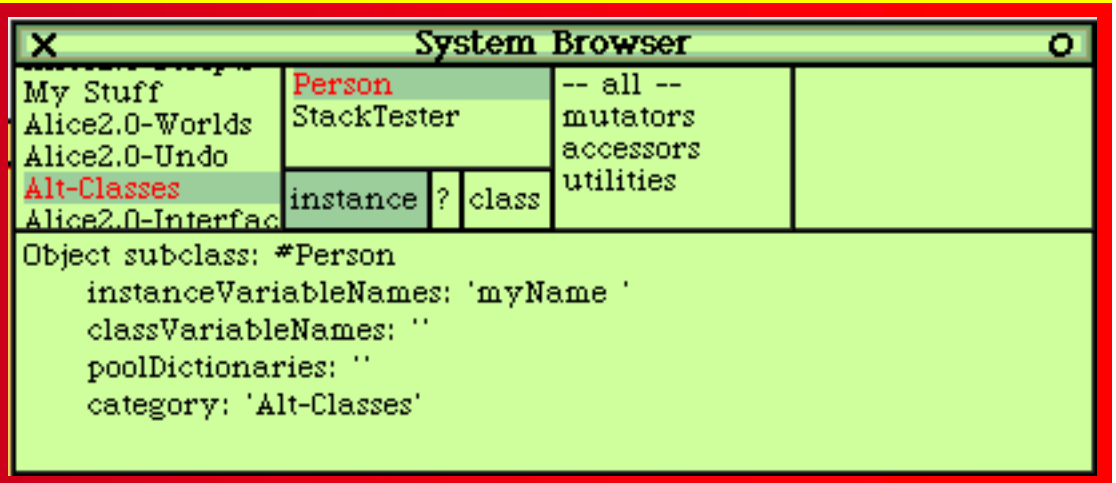

Use: Ada

```
Procedure payroll is
  EmpRef : EmployeeRef; fin: File_Type;
  eType: Character; Discard: Unbounded_String;
begin
  Open(fin, In_File, "payroll.dat");
  loop
    Get(fin, eType);  Discard := Get_Line(fin); // 'F', 'A', …
    exit when End_Of_File(fin);
    if empType = 'F' then EmpRef := new Faculty;
    elsif empType = 'A' then EmpRef := new Administrator;
    elsif empType = 'S' then EmpRef := new StaffMember;
    elsif empType = 'W' then EmpRef := new StudentWorker;
    end if;
    Get(EmpRef.all, fin);
    Put( GetName(EmpRef.all) ); New_Line;
    Put( GetPay(EmpRef.all) ); New_Line;
  end loop;
  close(fin);
// …
```

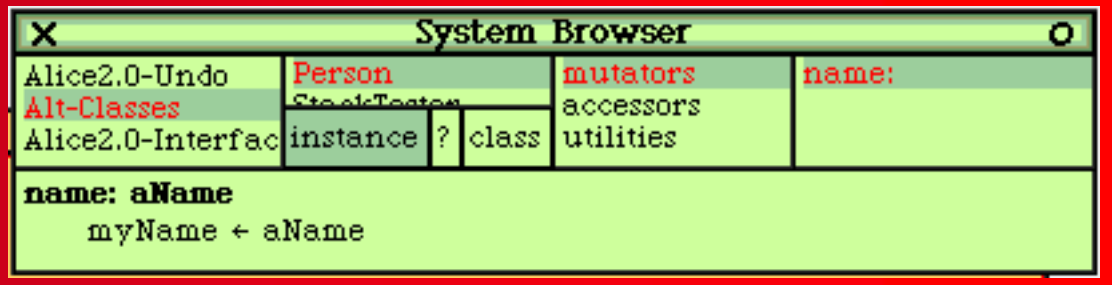
OO capabilities are an add-on in Ada, and they feel like it...

Implementation: Smalltalk

Smalltalk's GUI makes it easy to build our *Person* class:



We provide an *initialization* instance method:



Person inherits the *new* (class method) constructor from *Object*:

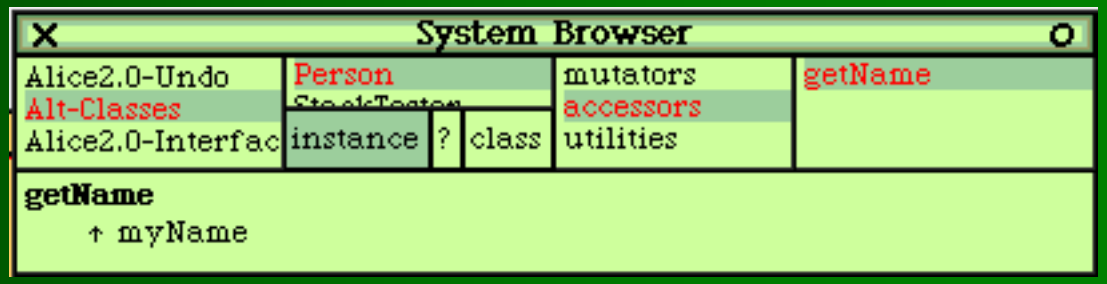
This allows us to write:

```
p := Person new name: 'Ann'.
```

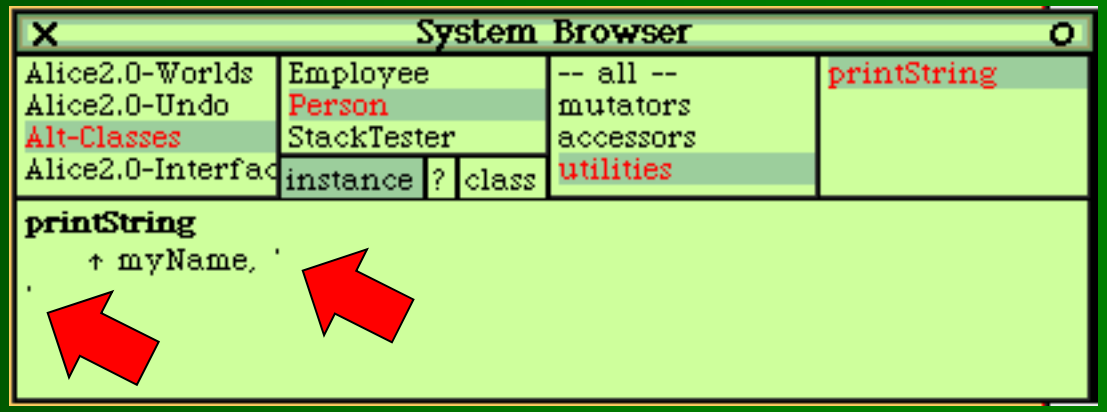
to *construct* and *initialize* a *Person*.

Implementation: Smalltalk (ii)

The *name accessor* is easy:



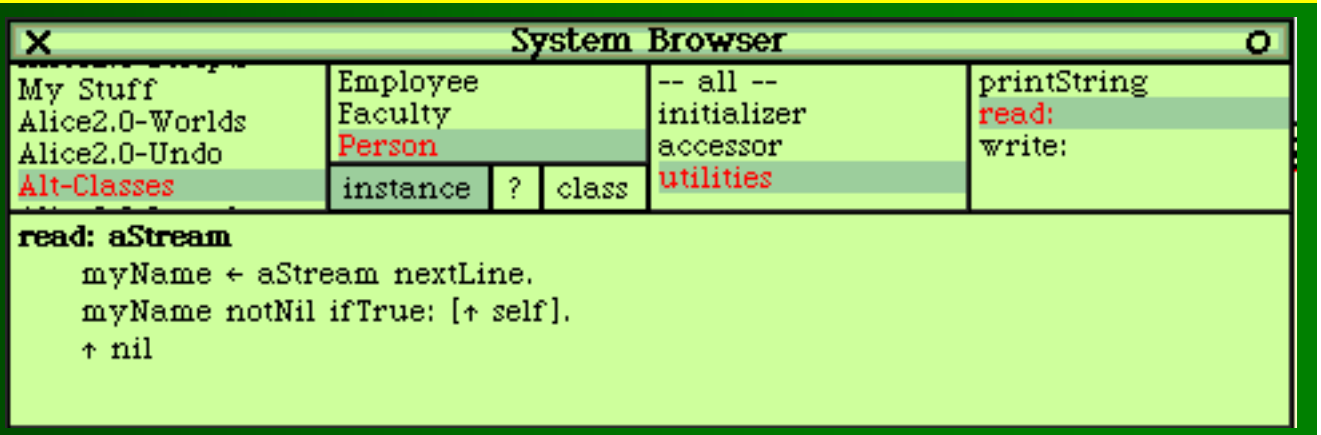
And to facilitate output, we define *printString* for a *Person*:



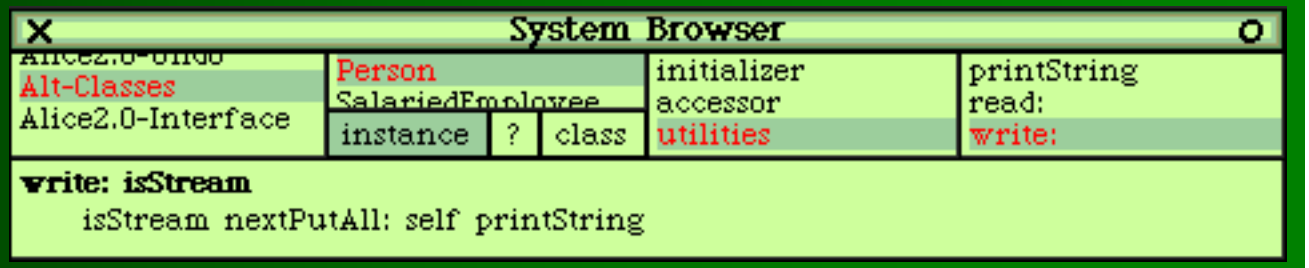
Note that Smalltalk allows strings to contain *embedded newlines*, which we use to separate *myName* from what follows it...

Implementation: Smalltalk (iii)

We might define a *read:* method as follows:



plus a *write:* method that uses *printString* to display itself:



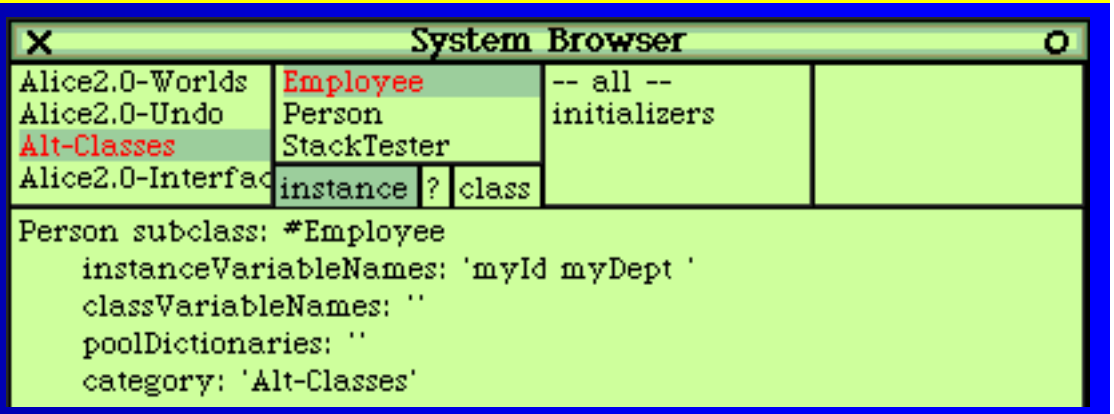
This lets us write:

```
p := Person new name: 'Ann'.
f := FileStream newFileNamed: 'data.txt'.
p write: f.
```

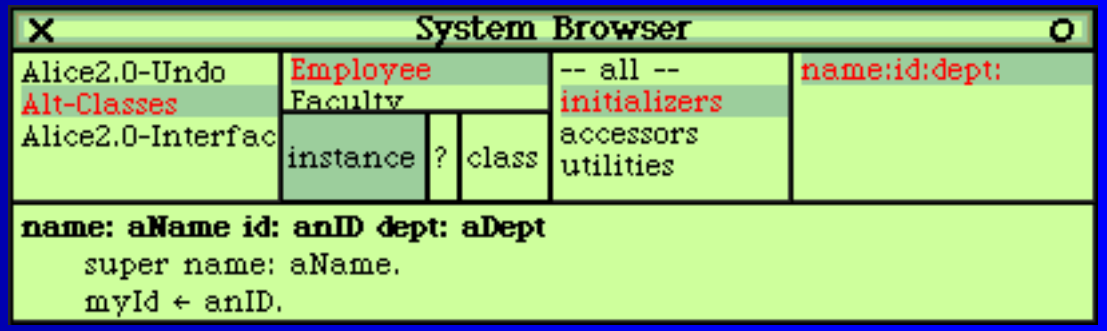
to create a stream to a file and write a *Person* to it.

Implementation: Smalltalk (iv)

We then build our *Employee* class as a subclass of *Person*:



plus a method to *initialize* an *Employee*:

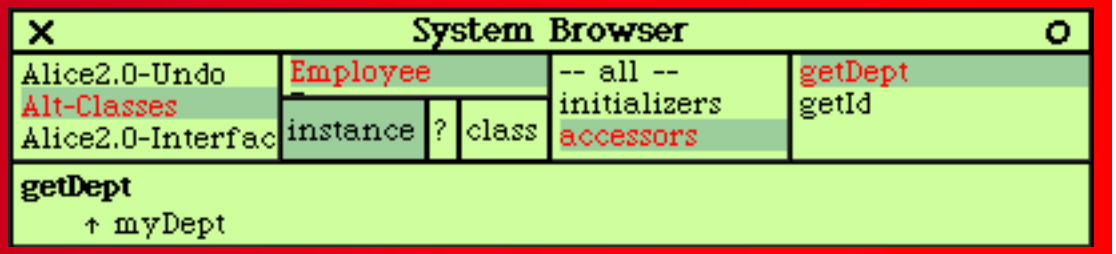


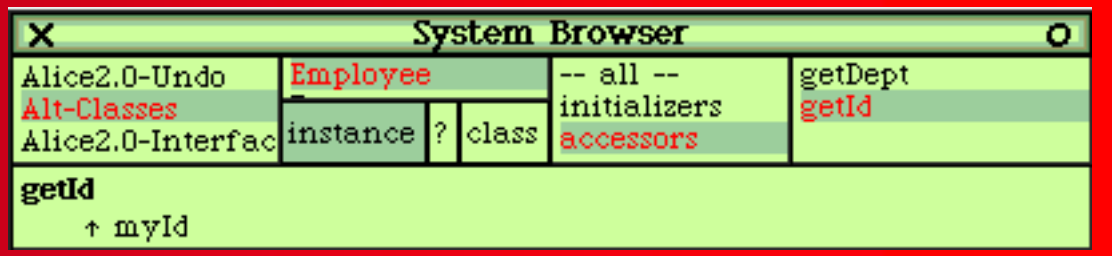
It is good practice to _____ (e.g., `super name: aName`) _____:

- It _____ we invested in writing those methods (avoid redundant code).
- If we alter the superclass method, the subclass _____.

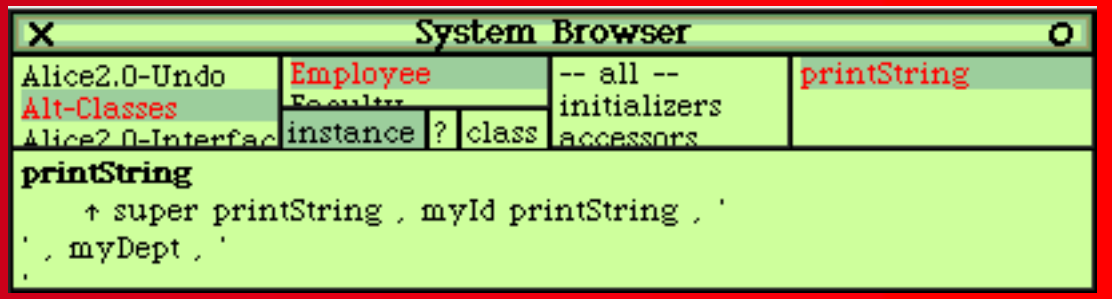
Implementation: Smalltalk (v)

We then define *accessors* for the instance variables:

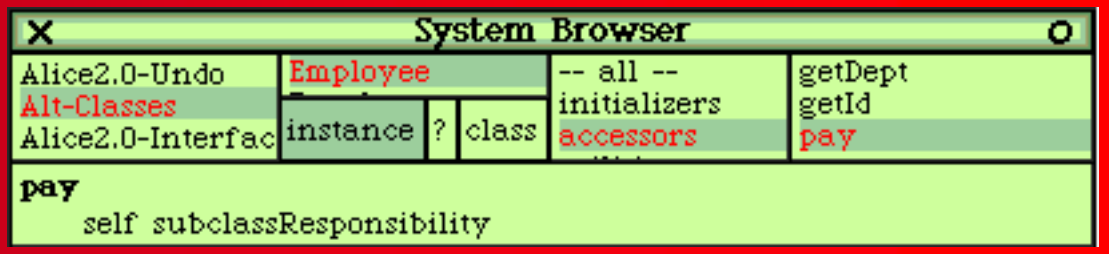




printString to facilitate output (using the superclass version):

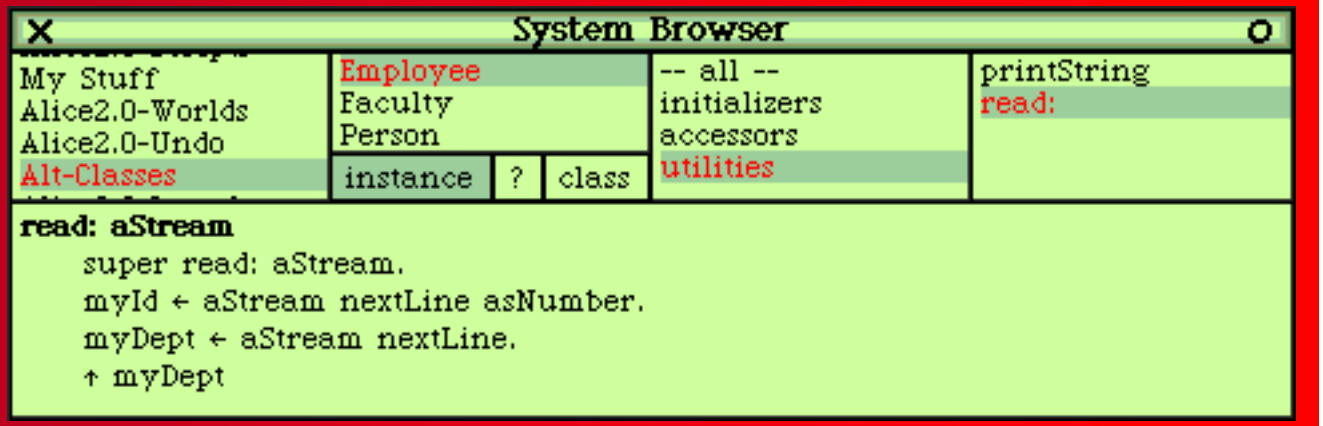


and *pay* as an abstract / "pure virtual" method:

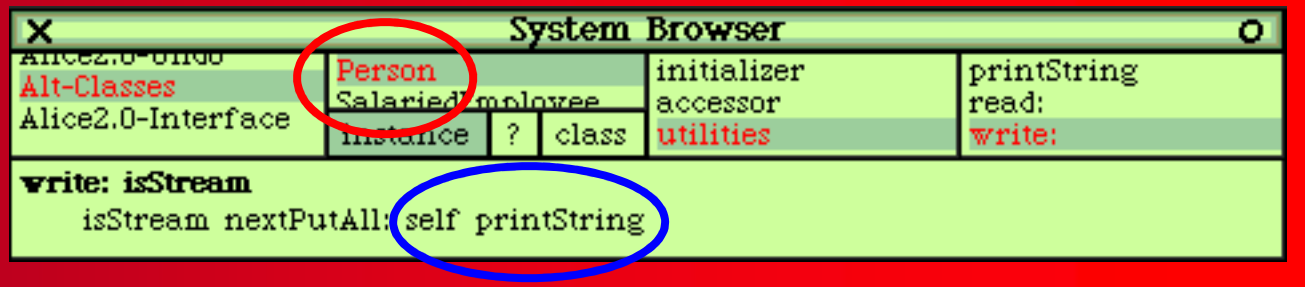


Implementation: Smalltalk (vi)

We can then override *read:* as follows:



Because (i) the *write:* we inherit from *Person* _____ _____ to display itself,



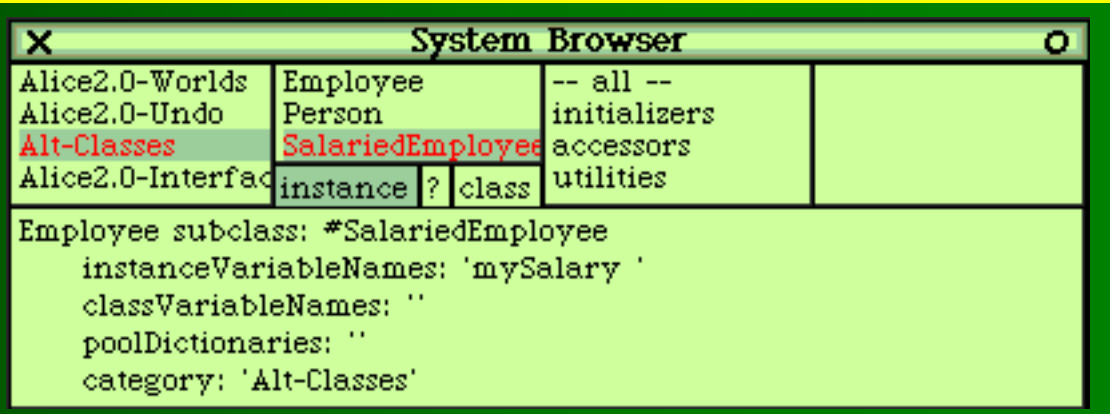
(ii) we have defined *printString* in class *Employee*, and

(iii) _____ Smalltalk methods are *polymorphic*,

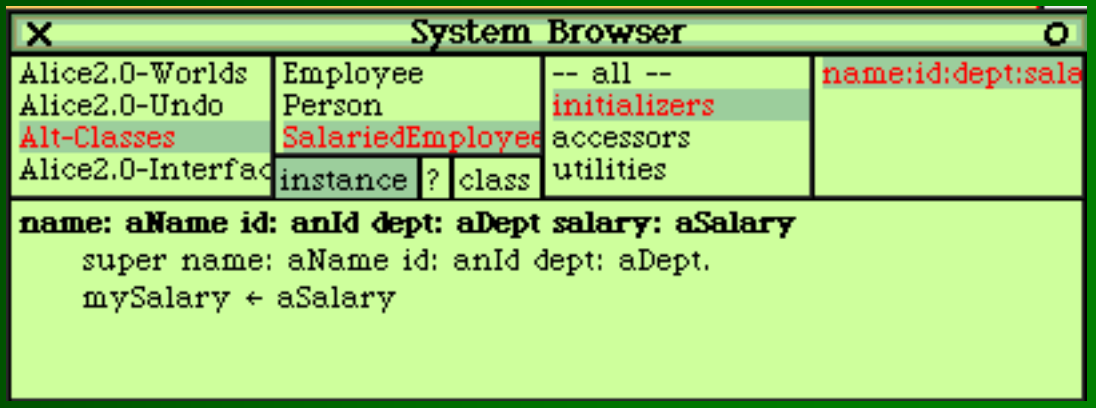
the *write:* we inherit from *Person* will correctly output an *Employee*...

Implementation: Smalltalk (vii)

We then build our *SalariedEmployee* class as a subclass of *Employee*:



plus a method to *initialize* an *SalariedEmployee*:



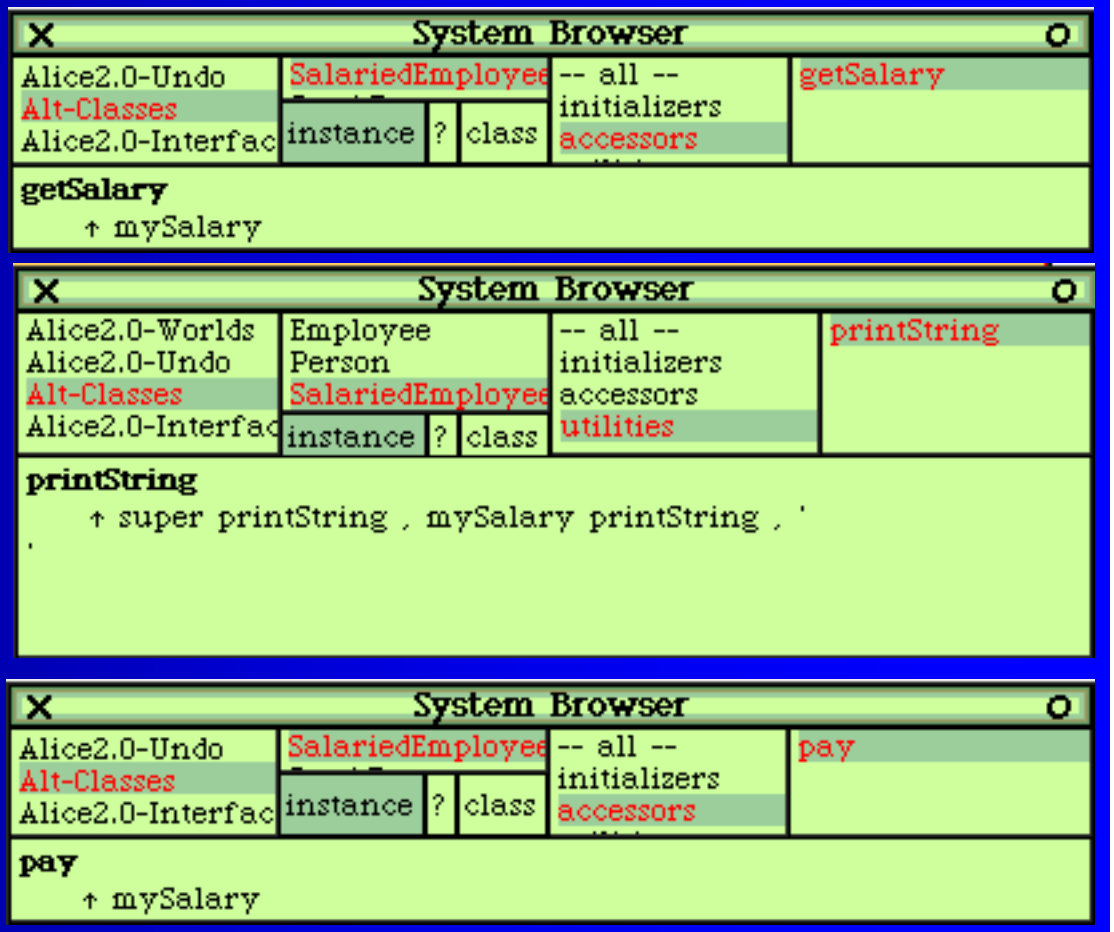
As before, we _____ and thus recoup the work we invested in writing them.

Implementation: Smalltalk (viii)

We then define an *accessor* for our instance variable:

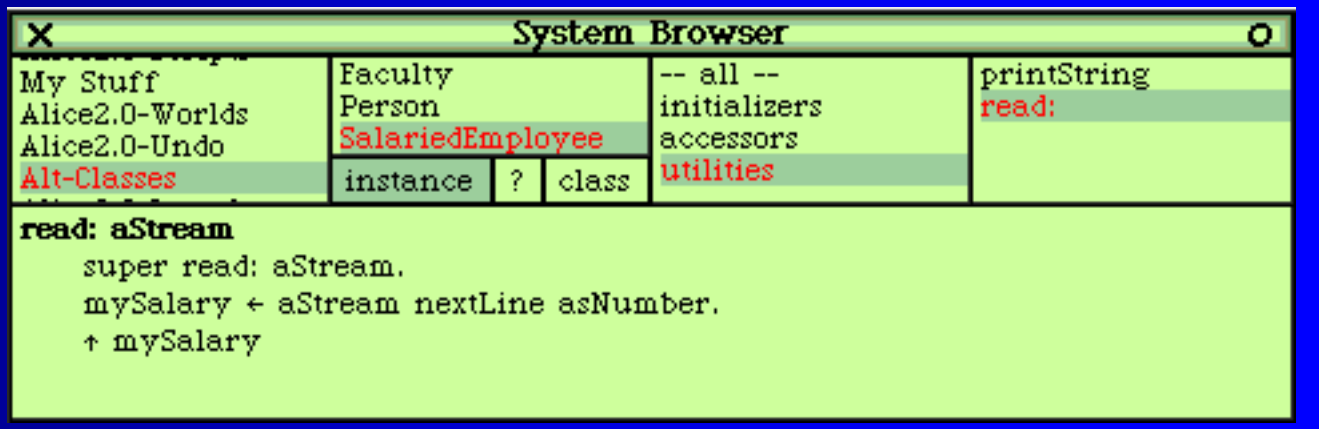
printString to facilitate output:

and the polymorphic *pay* method:



Implementation: Smalltalk (ix)

We can then override *read:* as follows:

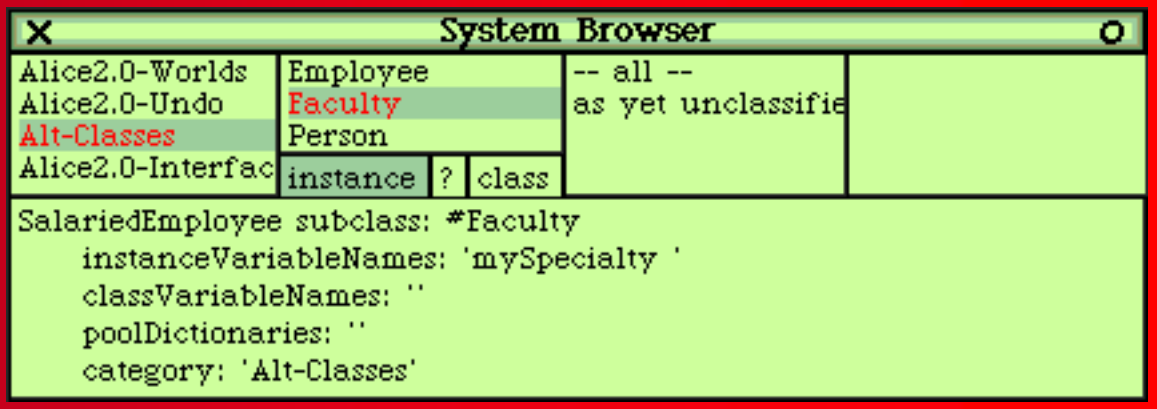


As before, our definition of (polymorphic) *printString* means that the *write:* we inherit from *Person* will correctly output a *SalariedEmployee* without any further work on our part.

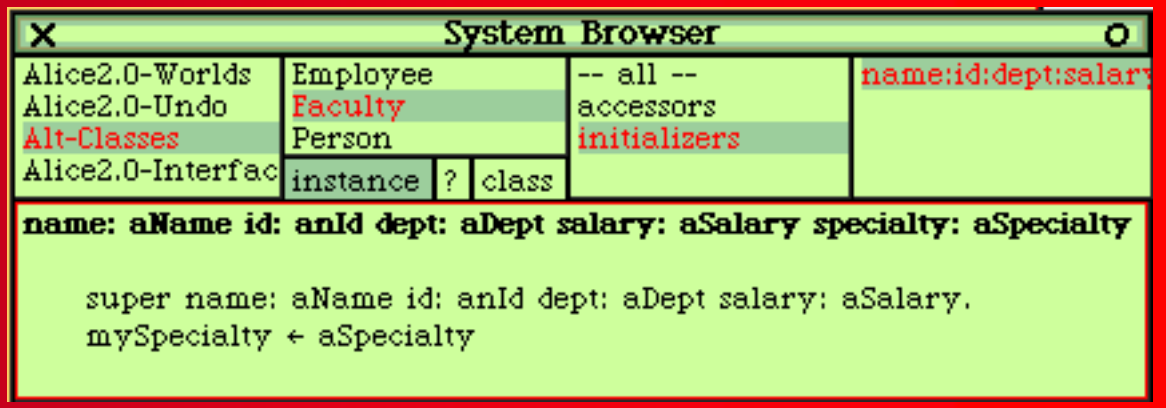
We could have performed input similarly, if we had defined *read:* in *Person* to use `self fromString`
and then defined *fromString* in *Person* and each of its subclasses.

Implementation: Smalltalk (x)

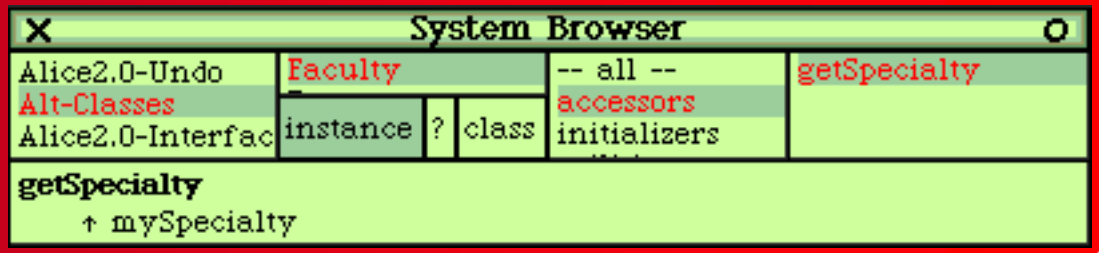
We then build our *Faculty* class as a subclass of *SalariedEmployee*:



plus methods to *initialize*:

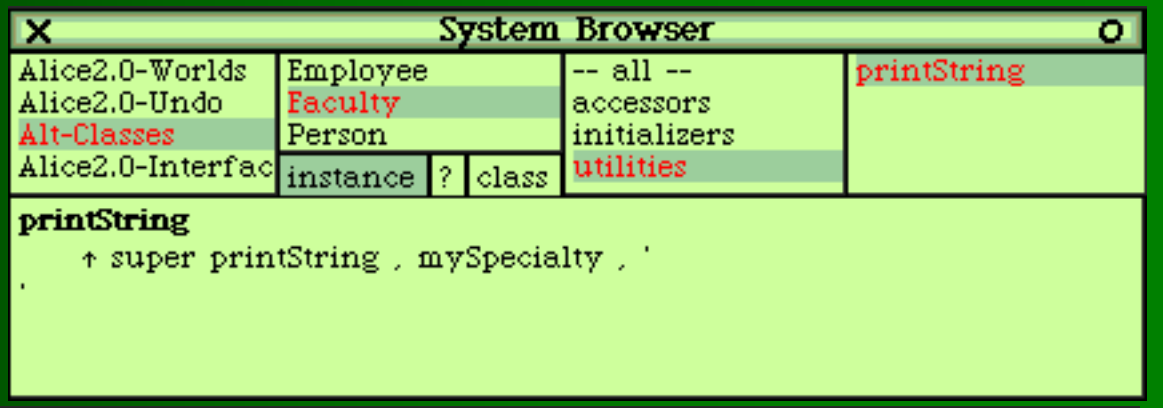


and *access* its instance variable:

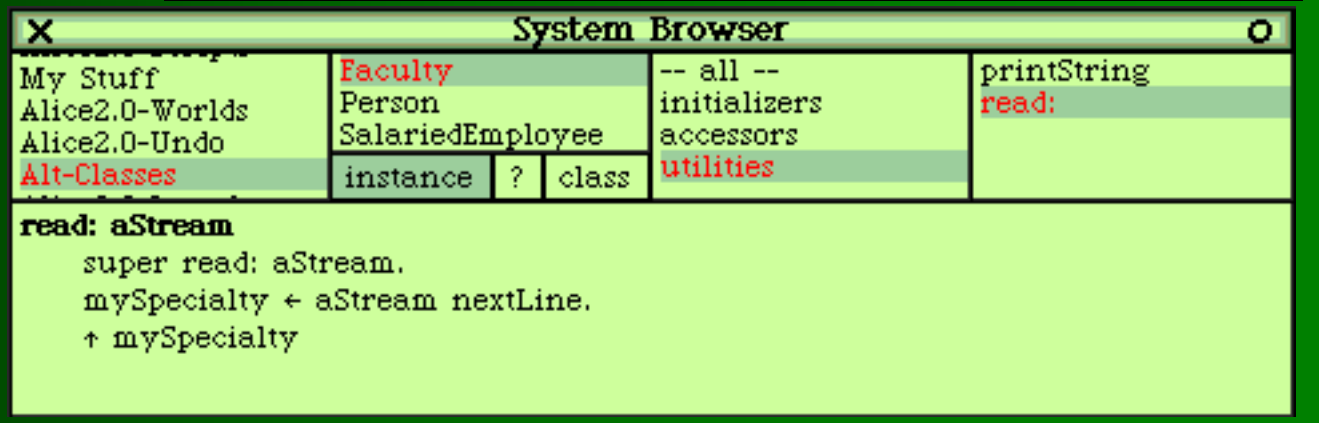


Implementation: Smalltalk (xi)

We then define *printString* to facilitate output:



and override *read:* to provide input:

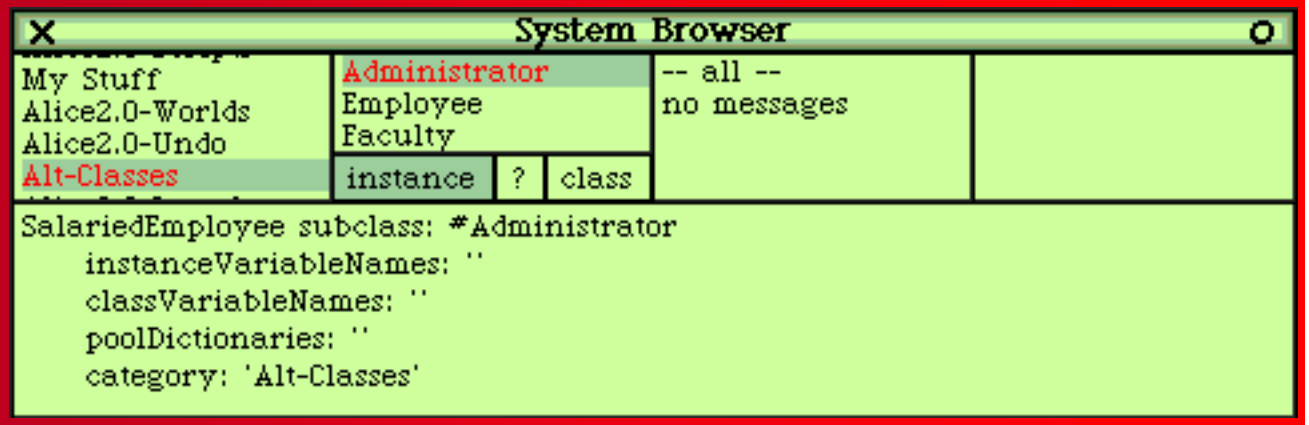


We then build the other classes in our design in a similar fashion, using inheritance and polymorphism to avoid redundant coding...

Implementation: Smalltalk (xii)

Administrator indicates how easy this is:

The class has no attributes beyond what it inherits from its superclass:



And since Smalltalk separates *construction* from *initialization*:

- Construction is via *new* inherited from _____;
- Initialization is via *name:id:dept:salary* inherited from _____;

we're done with *Administrator*!

Use: Smalltalk

Given the class hierarchy our design calls for, we can write something like this as the *run* method of a class that solves our payroll problem:

Workspace

```
| inFile emp empType |

inFile := FileStream oldFileNamed: 'test.in'.
[(empType := inFile nextLine) notNil]
  whileTrue: [
    empType = 'f'
      ifTrue: [ emp := Faculty new ]
      ifFalse: [
        empType = 'a'
            ifTrue: [emp := Administrator new ]
            ifFalse: [
                empType = 's'
                    ifTrue: [ emp := StaffMember new ]
                    ifFalse: [
                        empType = 'w'
                            ifTrue: [ emp := StudentWorker new ]
                            ifFalse: [ "... display an error-alert..." ] ] ] ].
    emp read: inFile.
    Transcript show: emp name; cr;
                show: (emp pay printString); cr
    ].
inFile close.
```


Summary

Object-oriented programming (OOP) is a way to build a system made up of a hierarchy of classes that reflects _____.

- A *subclass* inherits the _____ (data + operations) of its *superclass*.
- _____ ensures that when a message is sent to an object, the message is delivered to that object *first*:
  - If its class defines that message, that definition is invoked;
  - Otherwise, the message is sent "upward" in the hierarchy to the parent class, where the process is repeated.
  - If the message reaches the root class without finding a definition, a run-time error occurs.

This is called _____, because the same message: *handle msg* may produce very different behaviors, depending on the receiver.

Summary (ii)

"OO" languages differ in how easy/simple they make OOP:

language/ binding	Ada	C++	Java	Smalltalk
compile time (static)				
run time (dynamic)				

"OO" languages thus lie on an OO continuum:

less OO ← → more OO