

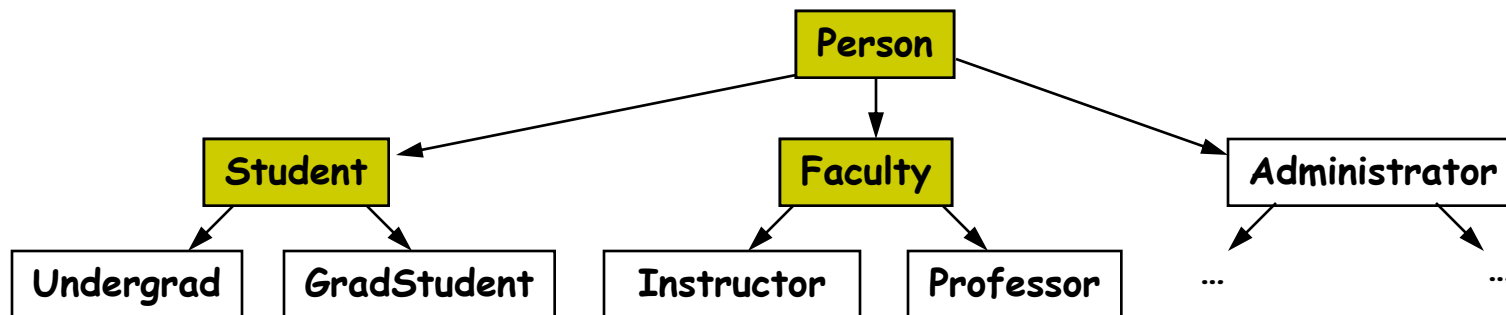
Implements vs. Extends When Defining a Class



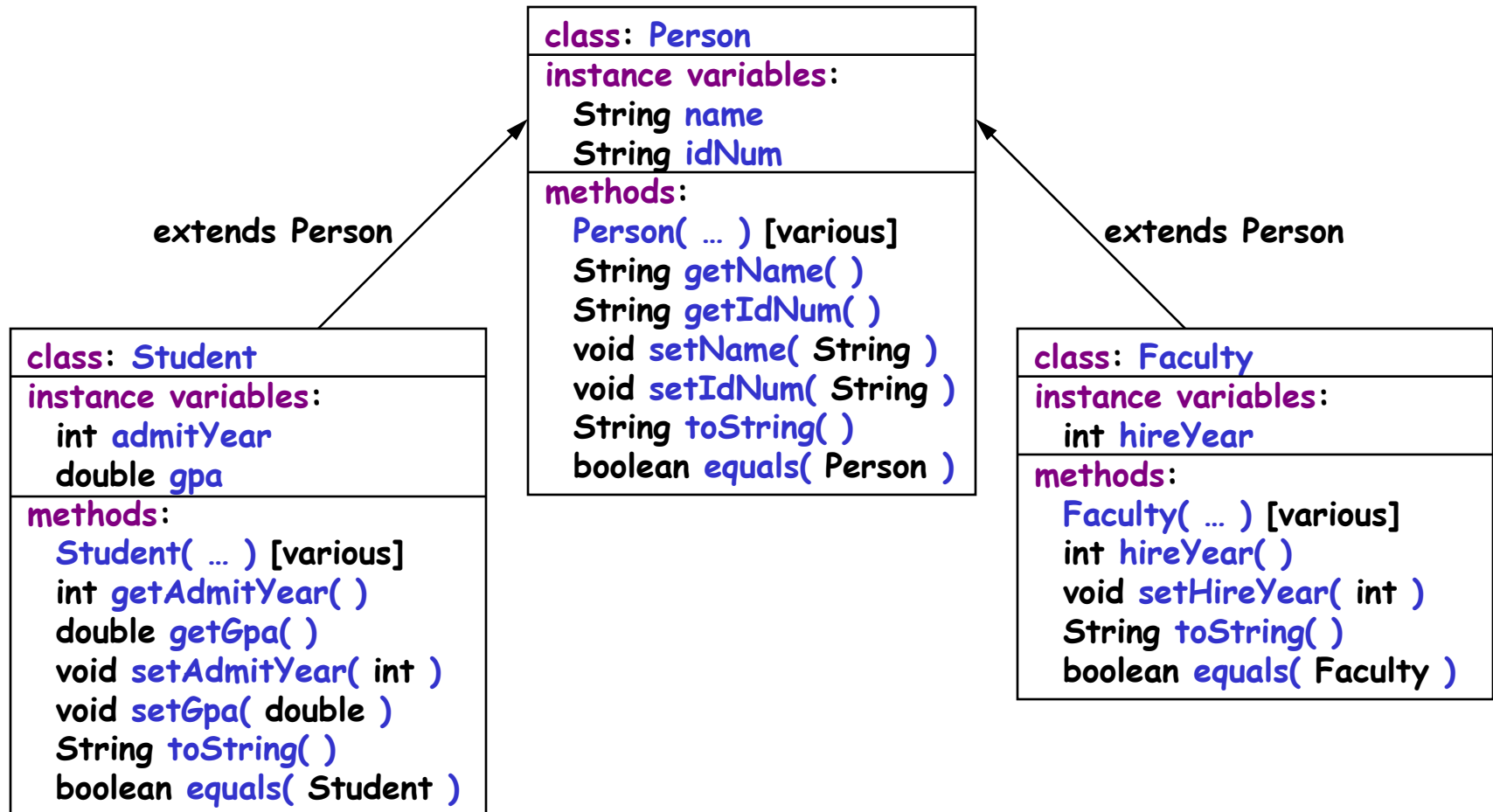
- **implements:**
 - Keyword followed by the name of an **INTERFACE**
 - Interfaces only have **method PROTOTYPES**
 - You **CANNOT** create an object of an interface type
 - Can have a reference of the interface type point to an object of the class that implements it
- **extends:**
 - Keyword followed by the name of a **CLASS**
 - That class contains full **method DEFINITIONS**
 - You **CAN** create objects of that base class type
 - Can have reference of the base class type point to an object of the class that extends it

Example: People at University

- Base class: person
- Derived classes: student, faculty, administrator
- Derived from those: undergrad, grad, instructor, professor, ...



University Person Example



Overriding vs. Overloading

- **Overriding**: a derived class defines a method with same name, parameters as base class
- **Overloading**: two or more methods have the same name, but different parameters
- Example

```
public class Person {  
    public void setName( String n ) { name = n; }  
    ...  
}  
public class Faculty extends Person {  
    public void setName( String n ) {  
        super.setName( "The Evil Professor " + n );  
    }  
    public void setName( String first, String last ) {  
        super.setName( first + " " + last );  
    }  
}
```

Base class setName()

Overriding

Overloading

Early vs. Late Binding

- Consider:

```
Faculty carol =  
    new Faculty("Carol Tuffteacher", "999-99-9999", 1995);  
Person p = carol;  
System.out.println( p.toString() );
```

- Which version of `toString` – `Person` or `Faculty` – is called?

- Early (static) binding

- `p` is declared to be of type `Person`
 - Therefore, the `Person` version of `toString` is used

- Late (dynamic) binding

- The object to which `p` refers was created as `Faculty` object
 - Therefore, the `Faculty` version of `toString` is used

- Java uses late binding (C++ by default uses early binding)

- Early binding is more runtime efficient (decisions about method versions can be made at compile time)
 - Late binding respects encapsulation (object defines its operations when it is created)

Polymorphism

- Java's **late binding** makes it possible for a single reference variable to refer to objects of many different types. Such a variable is said to be **polymorphic** (meaning having many forms).
- **Example**: Create an array of various university people and print.

```
Person[ ] list = new Person[3];  
list[0] = new Person( "Col. Mustard", "000-00-0000" );  
list[1] = new Student ( "Ms. Scarlet", "111-11-1111", 1998, 3.2 );  
list[2] = new Faculty ( "Prof. Plum", "222-22-2222", 1981 );  
for ( int i = 0; i < list.length; i++ )  
    System.out.println( list[i].toString( ) )
```

Output:

```
[Col. Mustard] 000-00-0000  
[Ms. Scarlet] 111-11-1111 1998 3.2  
[Prof. Plum] 222-22-2222 1981
```

- **What type is list[i]?** It can be a reference to any object that is derived from Person. The appropriate toString will be called.

Public, Protected, Package(default) and Private



- Select which level of visibility

Access Levels				
Access Level/Group	Class	Package	SubClass	World
public	Y	Y	Y	Y
protected _(avoid)	Y	Y	Y	N
package (default)	Y	Y	N	N
private	Y	N	N	N

Shadowing

- Can we override instance variables just like methods?
- Yes, but be careful!
 - Overriding instance variable is called **shadowing**
 - Shadowing hides instance variables of base class (can still access them using `super.varName` in subclass, but not in “outside world”)

```
public class Person {  
    String name;  
    ...  
}  
public class Administrator extends Person {  
    String name; // name refers to Administrator's name  
}
```

- Confusing! Better to pick a new variable name

Example of Overloading/Overriding



```
public class Base {  
    public void m (int x) { ... }  
}
```

```
public class Derived extends Base {  
    public void m (int x) { ... }  
    public int m (int x) { ... }  
    public void m (double d) { ... }  
}
```

Overriding: with increased visibility

Error! duplicate method declaration

Overloading

// The following appears in the same package as above

```
Base b = new Base( );  
Base d = new Derived( );  
Derived e = new Derived( );
```

calls Base:m(int)

calls Derived:m(int)

b.m (5);

d.m (6);

d.m (7.0);

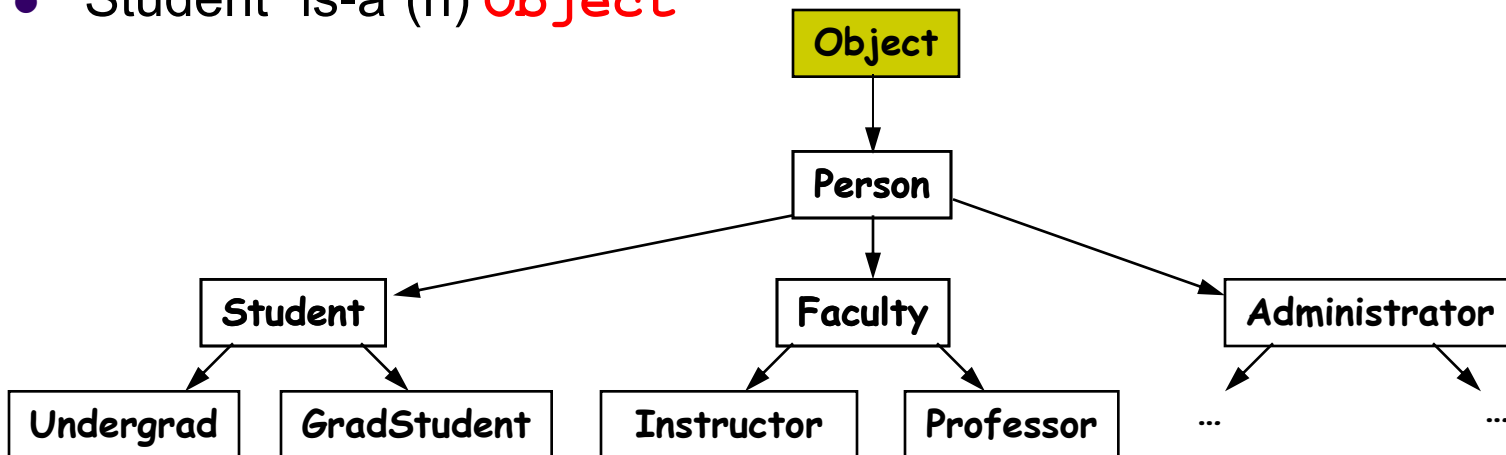
e.m (8.0);

Error! Since d is declared Base, the compiler looks for Base:m(double) Doesn't exist! So this does not make it past the compiler, even though Derived:m(double) is defined!

calls Derived:m(double)

Object

- Recall: inheritance induces “is-a” hierarchy on classes
 - Undergrad “is-a” Student
 - Student “is-a” Person
 - etc.
- Person “is-a”?
- Person “is-a”(n) **Object**
- Student “is-a”(n) **Object**



More on Object

- Special class at top of class inheritance hierarchy
- Defined in `java.lang` (so available in every program)
- Every class is derived (either directly or indirectly) from `Object`
 - If a class is not derived from anything, it is automatically derived from `Object`
 - e.g.

```
public class Foo { ...}
```

is equivalent to

```
public class Foo extends Object {...}
```
- Structure of `Object`
 - No instance variables
 - A number of methods, including:
 - `toString()`
 - `equals (Object o)`

Note: parameter to `equals` has type `Object`, so any object can be an argument

 - These methods can (and usually should) be overridden

Class vs. Type Information

- In Java
 - Every object is in one class (the one it was created from using `new`)
 - Objects may have many types (all those that class is based on)
 - Interfaces
 - Superclasses
- E.g. consider

```
Student bob = new Student();
Person p = bob;
```

 - Class of object pointed to by `bob` and `p` is `Student`
 - Type of object can be `Student`, `Person`, `Object`, etc.

Accessing Type Information

- **instanceof**

- Java boolean operator (not a method)
- Returns true if given object “is-a”(n) object of given (class) type
- E.g.

```
Student carol = new Student ( ... );  
if (carol instanceof Person) // true, because carol “is-a” Person
```

Object Casting

- Recall **casting** in primitive types
 - Casting: conversion of elements from one type to another
 - Widening Conversion
 - Every element in source type is a element in destination type
 - Can be done automatically

```
double x = 3; // 3 (int) widening conversion to double
```
 - Narrowing Conversion
 - Elements in source type are not necessarily elements in the destination type
 - Must use explicit type conversions to perform this casting

```
int x = (int)3.0; // 3.0 explicitly cast to int
```
- Similar notions can be found with object types also
 - Upcasting
 - Casting a reference to a **superclass** (casting up the inheritance tree)
 - Always done automatically and is always safe
 - Just ignore the parts that were added by the subclass
 - Downcasting
 - Casting a reference to a **derived** class
 - Requires explicit casting operator, which checks type info at run-time
 - Can cause runtime error

Safe Downcasting

- Illegal downcasting results in a thrown `ClassCastException` at run-time
- Q: Can we check for the legality of a cast before trying it?
- A: Yes, using `instanceof`
- Example
 - Given: `ArrayList` of university people
 - Want: Print the GPAs of the students
 - Solution approach
 - Iterate through list
 - Print GPAs only of Students

equals () Reconsidered

- Recall definition of equals ()

- ... in Person

```
public boolean equals (Person p) {  
    if (p == null){  
        return false;  
    }  
    return name.equals(p.getName()) &&  
        idNum.equals(p.getIdNum());  
}
```

- ... in Student

```
public boolean equals( Student s ) {  
    if (s == null){  
        return false;  
    }  
    return super.equals(s) &&  
        admitYear == s.admitYear &&  
        gpa == s.gpa;  
}
```

- What does following do?

```
public static void main (String[] args) {  
    Student bob = new Student ("R. Goode", "234-56-7890", 1998, 3.89);  
    Faculty bob2 = new Faculty ("R. Goode", "234-56-7890", 2005);  
    System.out.println (bob.equals (bob2));  
}
```

- true is printed!**

A Better equals ()

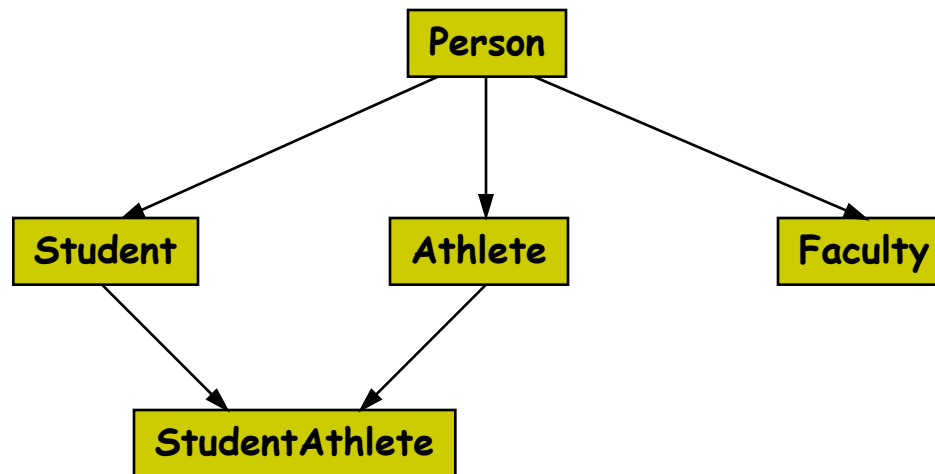
- Take Object as parameter
- Check for non-null-ness of parameter
- Check that class type is correct
- Then do other checks
- For example in Person:

```
public boolean equals (Object o) {  
    if ( ! (o instanceof Person)) {  
        return false;  
    }  
    Person p = (Person)o;  
    return name.equals(p.getName()) && idNum.equals(p.getIdNum());  
}
```

- Similar improvements can be made to Student, Faculty
- Now bob.equals(bob2) returns false

“Multiple Inheritance”?

- Intuitively useful to be able to inherit from multiple classes (**multiple inheritance**)

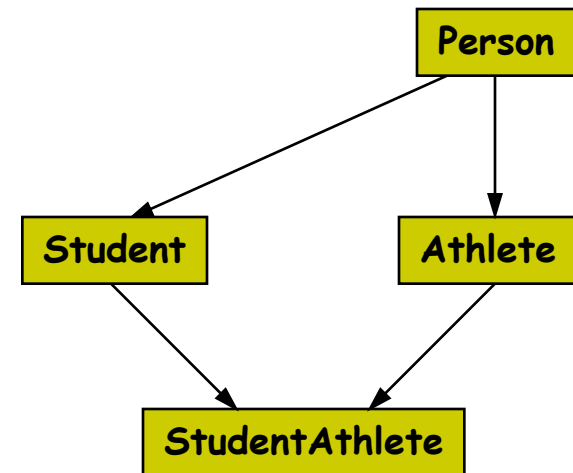


- But Java does not allow this

Why Does Java Disallow Multiple Inheritance?



- Semantic difficulties!
- Consider `StudentAthlete`
 - Objects would get name field from `Student`
 - Objects would also get name field from `Athlete`
 - Duplicate fields: what to do?
- Some languages (e.g. C++) do allow multiple inheritance



Can We Achieve Some of Benefits of Multiple Inheritance in Java?



- Yes, using interfaces + inheritance
 - Idea: use inheritance for one of inherited classes, interfaces for others
 - Interfaces ensure that relevant methods are implemented

- Example

```
public class Person { ... }
```

```
public class Student extends Person { ... }
```

```
public interface Athlete {  
    public String getSport ();  
    public void setSport (String sport);  
}
```

```
public class StudentAthlete extends Student implements Athlete {  
    ...  
}
```

- Objects of type `StudentAthlete` “are” `Students`
- They also can be wherever objects matching `Athlete` are required

Interfaces and Constants

- Interfaces can also contain `public final static` variables
- Sometimes interfaces are used to provide consistent definitions for constants throughout an application
- Example

```
public interface Months {  
    public final static int    JANUARY = 1;  
    public final static int    FEBRUARY = 2;  
    public final static int    MARCH = 3;  
    ...  
    public final static int    DECEMBER = 12;  
}  
  
public class MonthDemo implements Months {  
  
    public static void main( String[ ] args ) {  
        System.out.println( "March is month number " + MARCH );  
    }  
}
```

Because `MonthDemo` implements `Months`, it has access to the constants