Relationships Between Objects

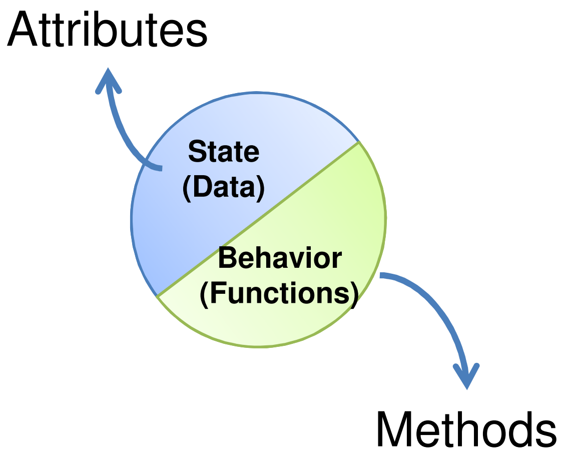
Object Oriented Programming
2016375 - 5

Camilo López

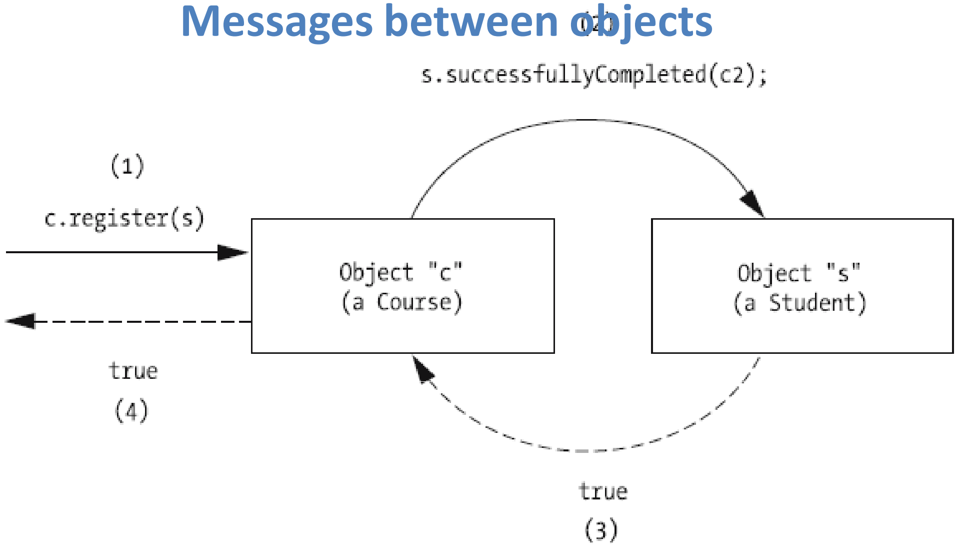
Outline

- Software Objects, Revisited
- Associations and Links
- Aggregation and Composition
- Inheritance

Software Objects, Revisited



Messages between objects



An object X is either temporarily handed a reference to object Y as an argument in a method call, or

temporarily requests a handle on Y from another object Z.

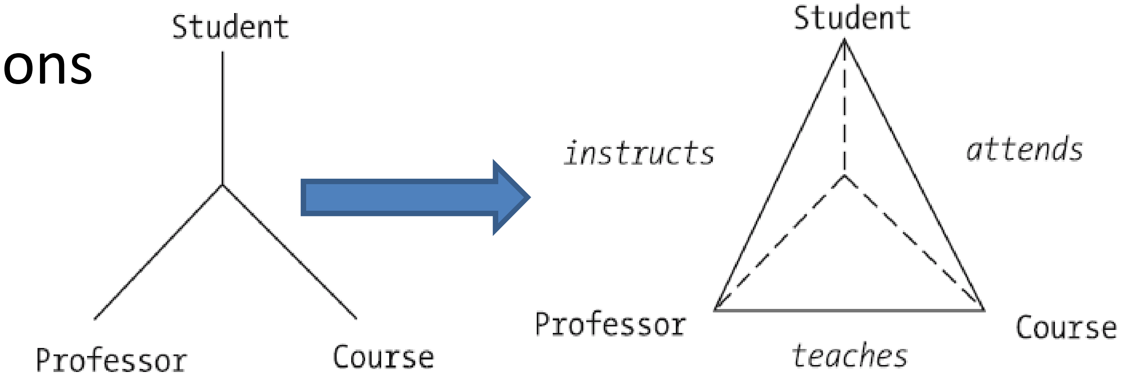
Behavioral relationships

Associations and Links

Association: ____________________ is enrolled in ____________________
(Some Student) (Some Course)

Link: _______Juan_______ is enrolled in _____OOP05_______
(A **specific** Student) (A **specific** Course)

- Associations enable links
- Binary/Unary associations
- Ternary associations
- Multiplicity
- Mandatory/optional



Associations and Links

Association: A Professor Department
A Department Professor

Association: A Professor Department
A Department Professors

Association: A Student Courses
A Course Students

Classes

Associations and Links

Association: A Professor chair Department
A Department have Professor (as chair)

Association: A Professor work for Department
A Department employ Professors

Association: A Student be enrolled in Courses
A Course have Students

Classes

Associations → Relationship

Associations and Links

Association: A Professor chair one Department
A Department have one Professor (as chair)

Association: A Professor work for one Department
A Department employ many Professors

Association: A Student be enrolled in many Courses
A Course have many Students

Classes

Associations → Relationship

Multiplicity: {(1:1),(1:m),(m:m)}

Associations and Links

Association: A Professor may chair one Department
A Department must have one Professor (as chair)

Association: A Professor must work for one Department
A Department may employ many Professors

Association: A Student may be enrolled in many Courses
A Course must have many Students

Classes

Associations → Relationship

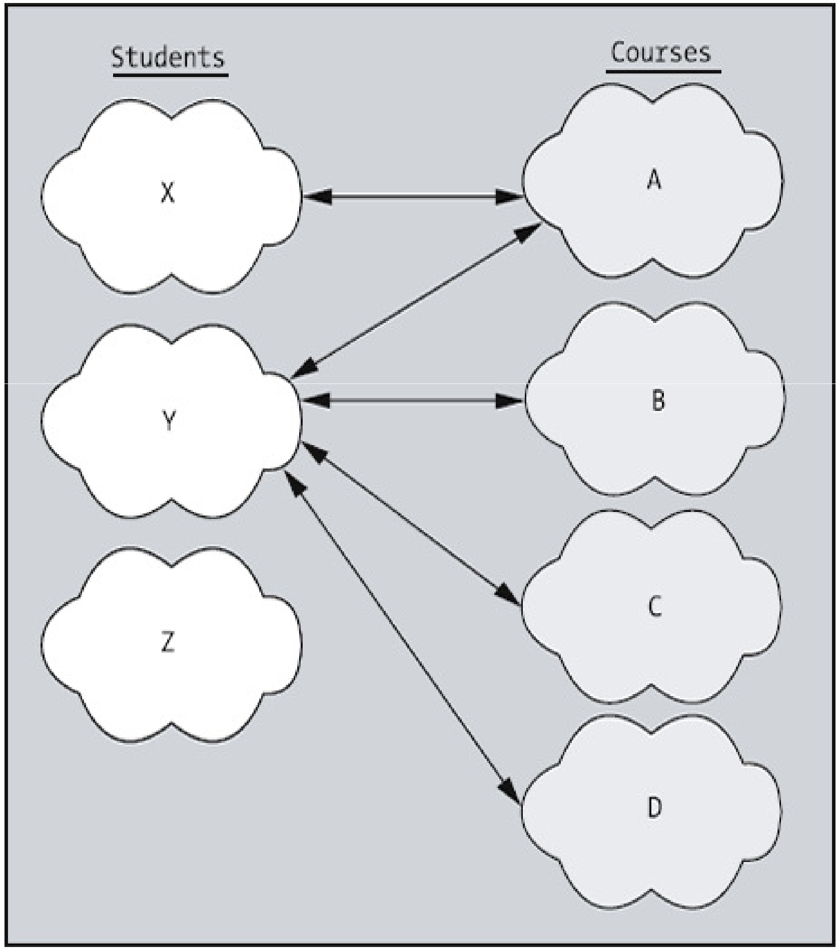
Multiplicity: {(1:1),(1:m),(m:m)}

Mandatory (must) / Optional (may)

↳ **At least one.**

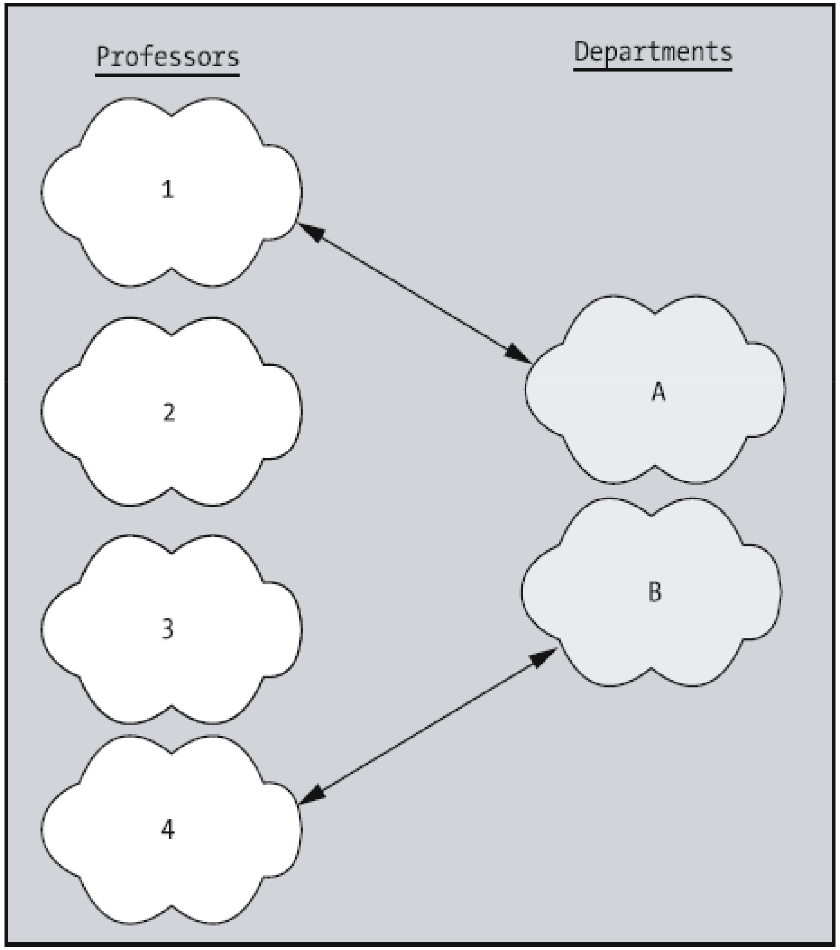
Associations and Links

many-to-many association



Associations and Links

one-to-one association



Aggregation and Composition

- **Aggregation** is a special form of association, alternatively referred to as the “consists of”, “is composed of” relationship.
  - For example, a car is composed of an engine, a transmission, four wheels, etc., so if Car, Engine, Transmission, and Wheel were all classes
  - A University ***is composed of*** *many Schools (the School of Engineering, the School of* Law, etc.).
  - A School ***is composed of*** *many Departments.*
- **Composition** is a strong form of aggregation, in which the “parts” cannot exist without the “whole.”
  - As an example, given the relationship **“a Book is composed of many Chapters”**, we could argue that a chapter cannot exist if the book to which it belongs ceases to exist.

Inheritance

- Let’s assume that we’ve accurately and thoroughly modeled all of the essential features of students via our Student class.

```
public class Student {
    private String name;
    // etc.

    public String getName(){
        return this.name;
    }
    public void setName(String name){
        this.name = name
    }
    //etc.
}
```

Inheritance

- Let’s assume that we’ve accurately and thoroughly modeled all of the essential features of students via our Student class.

```
public class Student {
    private String name;
    // etc.

    public String getName(){
        return this.name;
    }
    public void setName(String name){
        this.name = name
    }
    //etc.
}
```

NEW REQUIREMENTS!!

- What undergraduate degree the student previously received before entering his or her graduate program of study
- What institution the student received the undergraduate degree from

Inheritance

```
public class Student {
    private String name;
     private StringundergradDegree;
     private  String undergradInst;
    // etc.

    public String getName(){
        return this.name;
    }
    public void setName(String name){
        this.name = name
    }
    private  String getundergradStudent(){…}
    private  String setundergradStudent(String s){…}
    private  String getundergradInst(){…}
    private  String setundergradInst(String s){…}
    //etc.
}
```

NEW

REQUIREMENTS!!

- What undergraduate degree the student previously received before entering his or her graduate program of study
- What institution the student received the undergraduate degree from

Modify the Student Class

Inheritance

```
public class Student {
    private String name;
    private StringundergradDegree;
    private String undergradInst;
    private boolean gradStudent;
    // etc.

    public String getName(){…}
    public void setName(String name){…}
    private String getundergradStudent(){…}
    private String setundergradStudent(String s){…}
    private String getundergradInst(){…}
    private String setundergradInst(String s){…}
    private boolean isGradStudent(){…}
    private boolean setGradStudent(boolean g){…}
    //etc.
}
```

NEW REQUIREMENTS!!

- What undergraduate degree the student previously received before entering his or her graduate program of study
- What institution the student received the undergraduate degree from

Modify the Student Class

Inheritance

```
public class GradStudent {
    private String name;
     private StringundergradDegree;
     private String undergradInst;
    // etc.

    public String getName(){
        return this.name;
    }
    public void setName(String name){
        this.name = name
    }
    private String getundergradStudent(){…}
    private String setundergradStudent(){…}
    private String getundergradInst(){…}
    private String setundergradInst(){…}
    //etc.
}
```

NEW REQUIREMENTS!!

- What undergraduate degree the student previously received before entering his or her graduate program of study
- What institution the student received the undergraduate degree from

"Clone the Student Class"

Inheritance

public class SubClass **extends** SuperClass

```
public class GradStudent extends Student {
     private StringundergraduateDegree;
     private  String undergraduateInstitution;
    // etc.

   private  String getundergradStudent(){…}
   private  String setundergradStudent(){…}
   private  String getundergradInst(){…}
   private  String setundergradInst(){…}
    //etc.
}
```

NEW REQUIREMENTS!!

- What undergraduate degree the student previously received before entering his or her graduate program of study
- What institution the student received the undergraduate degree from

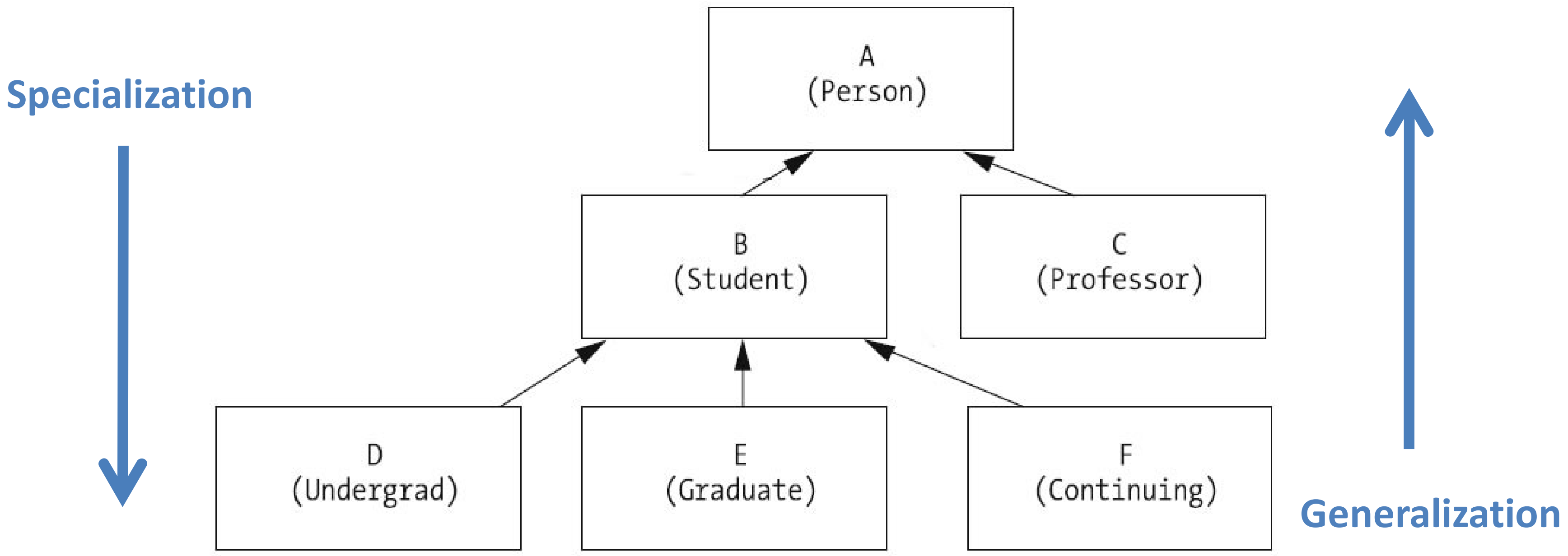
Taking advantage of Inheritance

Inheritance is often referred to as the "is a" relationship between two classes

Inheritance

An "acid test"

if there is something that can be said about a class A that can't be said about a proposed subclass B, then B really isn't a valid subclass of A.



Specialization

A (Person)

B (Student)

C (Professor)

D (Undergrad)

E (Graduate)

F (Continuing)

Generalization

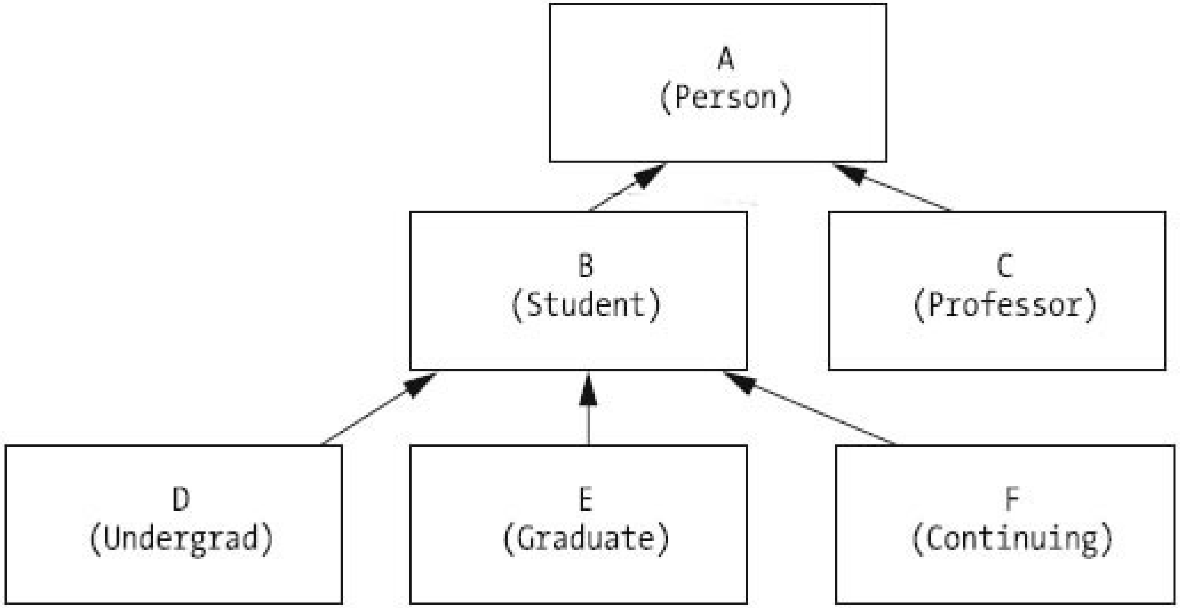
Inheritance

The Benefits of Inheritance

- Reduction of code redundancy.
  - Maintenance
  - Avoid “Ripple Effects”
- Subclasses are much more succinct than they would be without inheritance.
- Through inheritance, we can reuse and extend code that has already been thoroughly tested without modifying it.
- Best of all, we can derive a new class from an existing class even if we don’t own the source code for the latter!
- Classification is the natural way that humans organize information

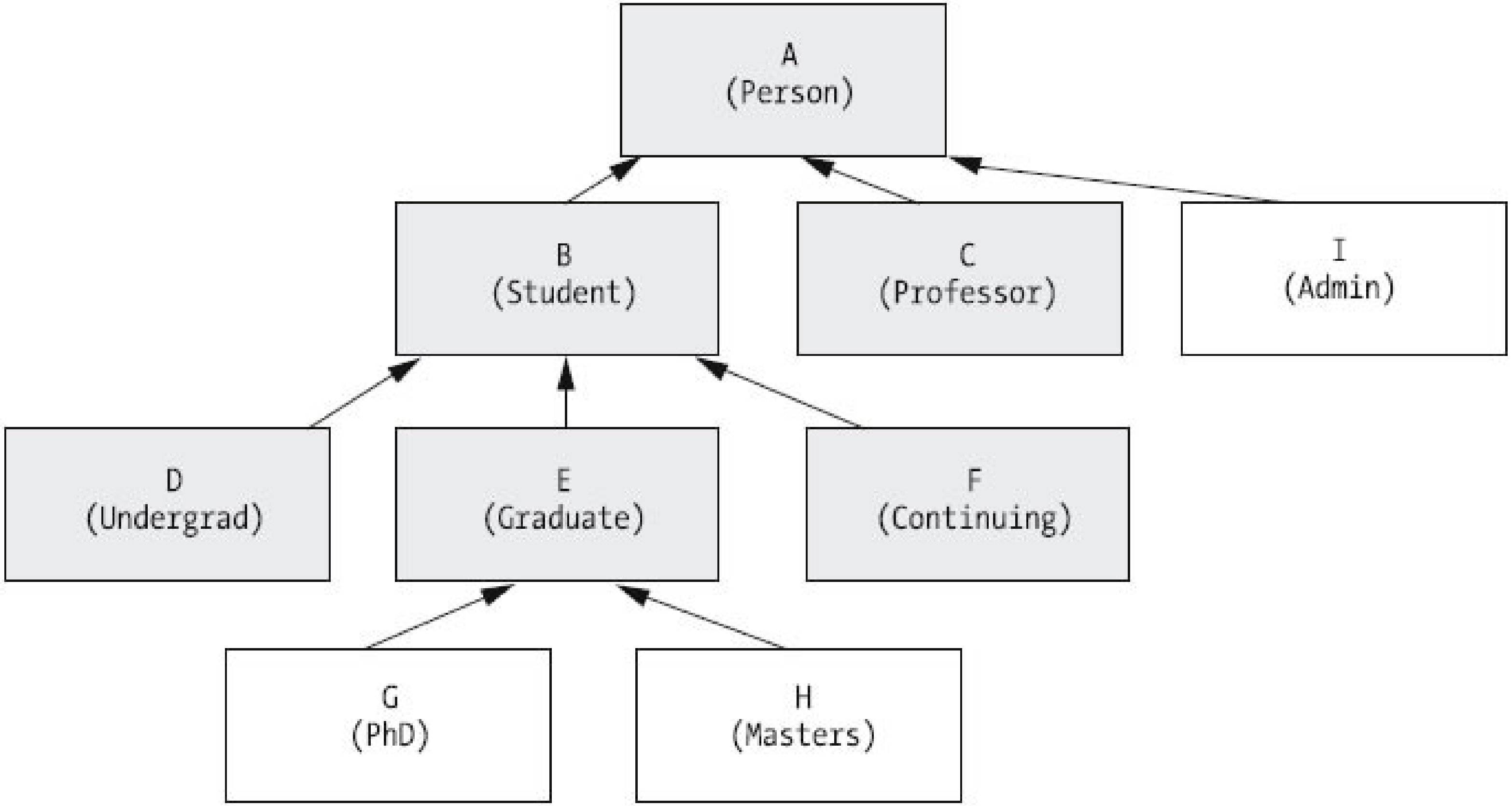
Inheritance

Class hierarchies inevitably expand over time.



Inheritance

Class hierarchies inevitably expand over time.



Inheritance

Deriving Classes

The Do’s

- ***extend*** the superclass by adding features.
- ***specialize*** *the way that a subclass performs one or more of the services* inherited from its superclass.

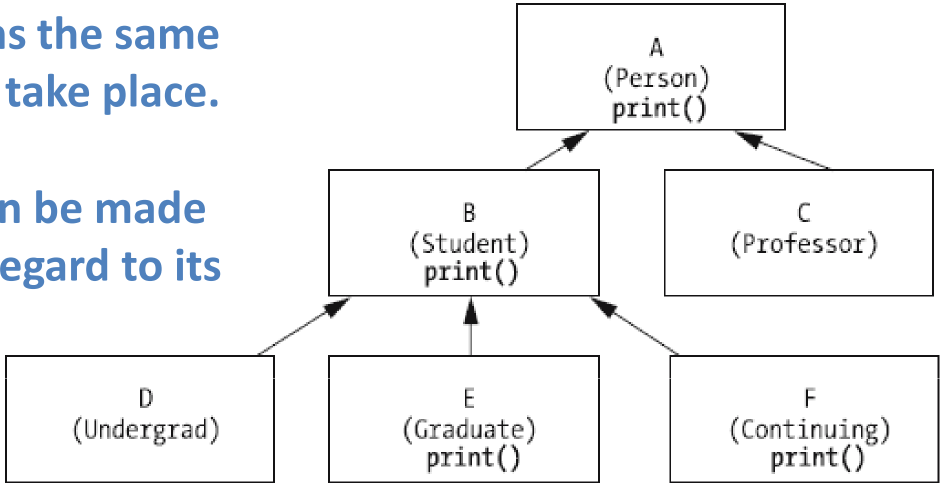
Specializing the way that a subclass performs a service—that is, how it responds to a given message as compared with the way that its superclass would have responded to the same message—is accomplished via a technique known as **overriding**.

Inheritance

Overriding

method signature remains the same -print()- for overriding to take place.

The only permitted change that can be made when overriding a method is with regard to its accessibility.



the accessibility granted to methodX() in the subclass cannot be more restrictive than the accessibility of the corresponding method in the superclass.

Any class not specifically overriding a given method itself will inherit the definition of that method used by its most immediate ancestor.

Inheritance

The “super” keyword

- Student version

```
public void print() {
    System.out.println("Student Name: " + this.getName() + "\n" +
                       "Student No.: " + this.getStudentId() + "\n" +
                       "Major Field: " + this.getMajorField() + "\n" +
                       "GPA: " + this.getGpa());
}
```

- GradStudent version

```
public void print() {
    System.out.println("Student Name: " + this.getName() + "\n" +
                       "Student No.: " + this.getStudentId() + "\n" +
                       "Major Field: " + this.getMajorField() + "\n" +
                       "GPA: " + this.getGpa());
                       "Undergrad. Deg.:" + this.getUndergradDegree() + "\n" +
                       "Undergrad. Inst.: " + this.getUndergradInst());
}
```

Inheritance

The “super” keyword

- GradStudent version

```
public void print() {
    System.out.println("Student Name: " + this.getName() + "\n" +
                       "Student No.: " + this.getStudentId() + "\n" +
                       "Major Field: " + this.getMajorField() + "\n" +
                       "GPA: " + this.getGpa());
                       "Undergrad. Deg.:" + this.getUndergradDegree() + "\n" +
                       "Undergrad. Inst.: " + this.getUndergradInst());
}
```

- GradStudent version using the “super” keyword

```
public void print() {
    super.print();

    System.out.println("Undergrad. Deg.:" + this.getUndergradDegree() + "\n" +
                       "Undergrad. Inst.: " + this.getUndergradInst());
}
```

Reuse code by calling the print method as defined by the Student superclass

Inheritance

The “super” keyword

```
public class Subclass extends Superclass {
    public void foo(int a, int b) {
        super.foo(a, b);
    }
}
```

Passing the argument values a and b through to our superclass's version of foo

```
public class Subclass extends Superclass {
    // We're overriding the foo method.
    public void foo(int a, int b) {
        int x = 2; // a local variable
        super.foo(a, x);
    }
}
```

Using selected argument values through to our superclass‘s version of foo

Inheritance

The “super” keyword

```
public class Subclass extends Superclass {
    // We're overriding the foo method.
    public void foo(int a, int b) {
        int x = 2; // a local variable
        super.foo(x, 3);
    }
}
```

Here, we're using neither a nor b as an argument.

```
public class Subclass extends Superclass {
    // We're overriding the foo method
    public int foo(int a, int b) {
        int x = 3 * a;
        int y = 17 * b;
        return super.foo(x, y);
    }
}
```

Assuming that foo was declared with an int return type in the superclass

Inheritance

Deriving Classes

The Don'ts

- We **shouldn't change the semantics**—that is, the intention, or meaning—of a feature.
  - Student (superclass): print() →display the values of all of an object's attributes in the command window
  - GraduateStudent (subclass): print() → it directs all of its output to a file instead
- We can't physically eliminate features, nor should we effectively eliminate them by ignoring them.
  - To attempt to do so would break the spirit of the "is a" hierarchy.
  - If a GraduateStudent could eliminate an attribute that it inherits from Student, for example, is a GradStudent ***really*** a Student after all?
  - Disable a method by overriding it with a "do nothing" version

Inheritance

Private Features and Inheritance

```
public class Person {
    <accessibility modifier> int age;
    // Other details omitted.
}
```

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	N	N
private	Y	N	N	N

- private → And use accessor methods

Inheritance

Inheritance and Constructors

```
public class Person {
    String name;
    String ssn;

    public Person(String n, String s) {
        this.setName(n);
        this.setSsn(s)
    }
}
```

Inheritance

Inheritance and Constructors

```
public class Student extends Person {
    private String major;

    public Student(String n, String s) {
        this.setName(n);
        this.setSsn(s);
        this.setMajor(“UNDECLARED”);
    }

    public Student(String n, String s, String m) {
        this.setName(n);
        this.setSsn(s);
        this.setMajor(m);
    }
}
```

Inheritance

Inheritance and Constructors

```
public class Student extends Person {
    private String major;

    public Student(String n, String s) {
        super(n,s);
        this.setMajor(“UNDECLARED”);
    }

    public Student(String n, String s, String m) {
        super(n,s);
        this.setMajor(m);
    }
}
```

if the super(…) **syntax is used, the call must be the first statement in the subclass constructor**

Inheritance

Inheritance and Constructors

```
public class Student extends Person {
    private String major;
    //…
    public Student(String m) {
        this.setMajor(m);
    }
}
```

No explicit call to super(args)

```
public class Student extends Person {
    private String major;
    public Student(String m) {
        super();
        this.setMajor(m);
    }
}
```

This could bring a compiler ERROR!!! ⚠

then, super() **is implied**

References

- J. Barker, *Beginning Java Objects: From Concepts To Code, Second Edition*, Apress, 2005.
- Java SE Tutorials (Last Updated 5/27/2009), which can be found at: http://java.sun.com/docs/books/tutorial