PART 8: EXTENDING CLASSES

Main topic: how to create a new class of objects by taking an existing class and adding new features. New class is said to *extend* old class.

One of key ideas in object oriented programming.

- Simple subclasses.
- Protected access.
- The staff-student problem.

(Horstmann Chapter 9)

Simple subclasses

Java, along with all other OO languages, supports idea of one class being a *subclass* of another. This enables Java programs to accurately represent those situations in the real world where all instances of one sort of entity are also instances of another (more general) sort.

Eg: all monkies are mammals; all medical students are students; all Ford Pintos are vehicles, etc.

Going back to the *Student* class from part 3, there are extra, specialised kinds of student, such as *PlacementStudent* which need additional information (such as the organisation they are doing their placement at) and methods to modify/read this information.

Simple subclasses

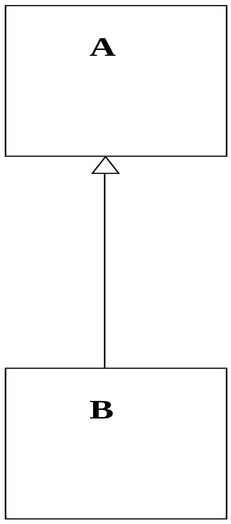
In general, suppose we have class A, and we want to define new class, B, whose objects have all features of objects of A, plus additional features. Can define B like this:

```
public class B extends A
{  Additional attributes, static variables,
   instance methods, static methods,
   and constructors.
}
```

B is said to *extend* A. Also, B is called a *subclass* of A, and A is called a *superclass* of B.

This situation is denoted by a class diagram like the following.

Inheritance of A by B



B objects will have all features (variables, methods and constructors) given in definition of A. Said to *inherit* features of A.

Can define new methods of B objects, and give new definitions for some old methods of A. The new method is said to *override* old one. Not usually sensible to do this for instance/static variables.

B may have its own constructors.

In effect, a B object has dual personality. Can be used strictly as a B object, e.g. one of new methods from definition of B can be used with it. Or can be used as special case of an A object. Java will allow any code written for A objects to be applied to B objects. This feature of Java is known as *polymorphism.*

Since Java is willing to treat B objects as if they are A objects, can think of B objects as if they actually belong to class A. It is for this reason that B is called a *subclass* of class A.

Placement students

```
class PlacementStudent extends Student
{ private String placedAt = "";

  public PlacementStudent(String id, String nm, String deg)
  { super(id,nm,deg); }  /* Invokes Student constructor */

  public void setPlacement(String org)
  { placedAt = org; }

  public void display()
  { super.display();   /* invokes display method of Student */
    System.out.println("Placement at: " + placedAt);
  }
}
```

PlacementStudent has attributes *idNumber*, *name*, *degree*, and *year* inherited from *Student*. Cannot be directly referred to in *PlacementStudent* because declared *private* in *Student*. Plus it has its own attribute *placedAt*.

PlacementStudent has methods *setDegree*, *advanceYear*, *getName* and *display* inherited from *Student*. These were *public* in *Student* so can be referred to within *PlacementStudent*, and we can use *Student* methods on *PlacementStudent* objects: *pstud.advanceYear*(), etc where

```
PlacementStudent pstud =
  new PlacementStudent("991889","yannis","CS/Year in Industry");
```

Plus it has its own methods *setPlacement* and *display* (redefined from *Student* – this is called ‘overriding’ the old *display* method).

Can set *pstud* placement by *pstud.setPlacement*(“*accenture*”), etc.

Inheritance hierarchies

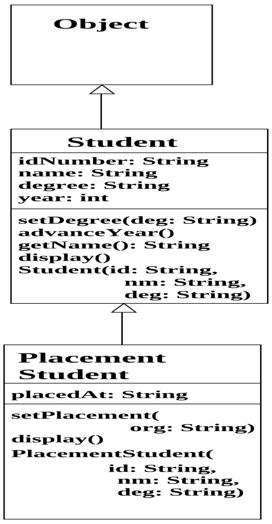
Every class inherits a class called *Object* which contains default definitions of *equals*(*Object obj*), *toString*(), etc, applicable to any class.

Override these in your own classes to provide more specific versions. Eg:

```
public boolean equals(Object obj)
{ if (obj instanceof Student)
  { Student other = (Student) obj;
    if (other.idNumber.equals(idNumber))
    { return true; }
  }
  return false;
}
```

for *Student*.

Inheritance Hierarchy



Protected Access

idNumber is private in *Student*, so cannot be accessed in *PlacementStudent*.

This gives designer of *Student* class freedom to make changes to any private features, e.g. to improve performance of the code, so long as the public features perform according to specification. If another programmer produces an extension to the original class that depends on these private features, and if the designer modifies the original class, the new extended class may suddenly cease to work correctly.

However, Java allows designer of a class to give its members a level of access in between private and public. Members can be declared to be `protected`. This means they can be accessed from any class that extends original class, but not by other classes.

If change *idNumber* declaration in *Student* to:

```
protected String idNumber;
```

can define

```
public boolean equals(Object obj)
{ if (obj instanceof PlacementStudent)
  { PlacementStudent other = (PlacementStudent) obj;
    if (other.idNumber.equals(idNumber))
    { return true; }
  }
  return false;
}
```

in *PlacementStudent*.

Alternatively provide accessor methods to return values of private attributes.

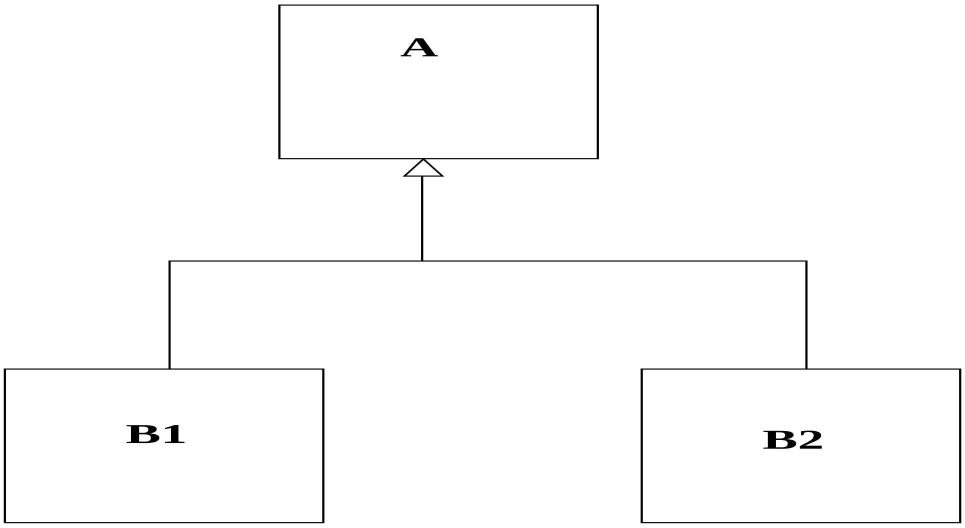
The Student-Staff Example

We've covered situation where have a class A, and extend it to produce subclass B whose objects consist of A objects with additional features.

This section describes another situation where it is useful to extend a class: when two different classes of objects, $B1$ and $B2$ have common properties, and need to be processed in similar ways. For example, you might want to create a list of objects made up of mixture of $B1$ objects and $B2$ objects, and call the same methods on all the list objects.

To do this, define superclass A of both $B1$ and $B2$.

Superclass A of $B1$ and $B2$



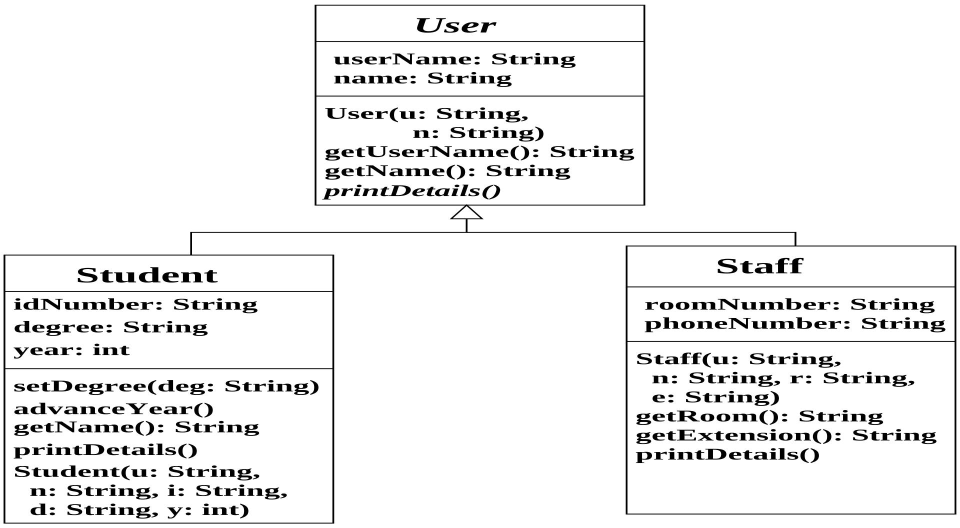
Defining a Common Superclass

The common operations applicable to both $B1$ and $B2$ objects can be programmed as methods of A.

Since $B1$ objects and $B2$ objects are going to be treated alike at times, is quite likely they will have some attributes and methods in common. All these common features can be defined in A.

Example: classes that might be used in a simple database system that records users of Department's computer systems. Two kinds of user: student users and staff users. Classes called *Student* and *Staff*. Superclass that contains Student and Staff objects is called *User*.

Class Diagram of Students and Staff



A *Student* object has following attributes:

1. String userName
2. String name
3. String idNumber
4. String degree
5. int year

The attributes for *Staff* objects are:

1. String userName
2. String name
3. String roomNumber

4. String phoneNumber

Last one is a four-digit telephone extension.

Student and *Staff* both have *username* and *name* attributes, so these can be moved to *User* class.

Each class will have own constructor to create a new object and initialise its attributes.

$User(u, n)$ constructor creates a *User* object and sets its username to u and name to n. Will be constructors $Student(u, n, i, d, y)$ and $Staff(u, n, r, p)$ for other two classes

Most of methods in this example are *accessor* methods, i.e. they return values of *getUserName* and *getName* of *User* will be inherited by both *Student* and *Staff*.

A method *printDetails* will display values of all data of object – so one version of *printDetails* for *Student* class, and different version in *Staff* class.

Also necessary to define *printDetails* in *User* class. A variable of type *User* can be declared – can hold either a *Student* object, or a *Staff* object:

```
User u = new Staff("gatesb", "Gates,Bill",
                   "3D", "2144");
u.printDetails();
```

Valid to assign a *Staff* object to variable of type *User* (since *Staff* is subclass of *User*).

For *u.printDetails*() to be valid, *User* must have a *printDetails* method. Define as *abstract* method:

```
abstract public void printDetails();
```

At run-time, when interpreter executes

```
u.printDetails();
```

will determine which subclass u belongs to, and will call *printDetails* method defined for that subclass.

Java selects right version of a method at run-time, depending on which subclass u belongs to – *dynamic binding*.

Since *User* class contains abstract method, it is an incomplete description of an object. So cannot create *User* objects – other than objects of *Student* or *Staff*. $new\ User(u, n)$ is invalid.

Use modifier `abstract`:

EXAMPLE 8.1 (User, Student and Staff classes)

```
/** A User object represents a user
    of the CS Department facilities. */

public abstract class User
{  private String userName;  // User's username.
   private String name;      // User's name.

   /** Create a User with username u and name n. */
   public User(String u, String n)
   {  userName = u;
      name = n;
   }

   /** Print details of the user. */
   abstract public void printDetails();
```

```
/** Return username of the user. */
public String getUserName()
{  return userName;  }

/** Return name of the user. */
public String getName()
{  return name; }

/** Program for testing User, Student and
    Staff classes.   */
public static void main(String[] args)
{ User u = new Staff("gatesb", "Gates,Bill",
                     "3D", "2715");
  u.printDetails();
  System.out.println();
  u = new Student("wonga", "Wong,Alice", "9912345",
                 "Maths and CS", 2);
  u.printDetails();
```



```
  }
}
```

The comments `/** ... */` are used by *javadoc* program to produce HTML documentation of Java programs.

```
/** A Student object represents a student user. */

class Student extends User
{  private String idNumber;
   // Student's ID number.
   private String degree;
   // Student's degree programme.
   private int year;
   // Student's year of study.

   /** Create new Student object
       with given username, name, ID number,
       degree programme and year. */
   public Student(String u, String n, String i,
                  String d, int y)
   { super(u,n);
     idNumber = i;
     degree = d;
```



```
     year = y;
   }

   /** Print details of the student. */
   public void printDetails()
   { System.out.println("STUDENT");
     System.out.println("Username: " + getUserName());
     System.out.println("Name: " + getName());
     System.out.println("ID number: " + idNumber);
     System.out.println("Degree: " + degree);
     System.out.println("Year: " + year);
   }

   /** Return student's ID number. */
   public String getIdNumber()
   { return idNumber; }

   /** Return student's degree progamme. */
```

```
   public String getDegreeProg()
   { return degree; }

   /** Return student's year. */
   public int getYear()
   { return year; }
}
```



```
/** A Staff object represents a member of staff
    who is a user. */

class Staff extends User
{  private String roomNumber;    // Staff member's room number.
   private String phoneNumber;   // Staff member's extension.

   /** Create new Staff object
       with given username, name,
       room number and extension.  */
   public Staff(String u, String n,
                String r, String e)
   { super(u,n);
     roomNumber = r;
     phoneNumber = e;
   }

   /** Print details of member of staff. */
```

```
   public void printDetails()
   { System.out.println("STAFF");
     System.out.println("Username: " + getUserName());
     System.out.println("Name: " + getName());
     System.out.println("Room: " + roomNumber);
     System.out.println("Ext: " + phoneNumber);
   }

   /** Return the staff member's room number. */
   public String getRoom()
   { return roomNumber; }

   /** Return staff member's extension. */
   public String getExtension()
   { return phoneNumber; }
}
```


Example 8.2

These three class definitions are all in single text file `User.java`. Generally each class should be in separate text file. But if several classes are put in single file should be closely related, and

1. only one class in file can be `public`, and
2. name of file must be same as that of the public class.

Any public class can have a main method with usual heading:

```
public static void main(String[] parameter-name)
```

Subclass Constructors

Think of each *Student* and *Staff* object as being built around a *User* object.

A constructor for a subclass, say *Student*, must start by constructing the superclass object at core of the subclass object. This is done by:

```
super(parameters);
```

where `parameters` must be list of parameters that matches parameters of one of superclass's constructors.

So constructors in *Student* and *Staff* both start with statement:

```
super(u,n);
```

where u and n are initial values for username and name.

Using a superclass method

Can save work in subclass methods like *printDetails* by putting common code in superclass:

```
public abstract class User
{  private String userName;
   // User's username.
   private String name;
   // User's name.

  public void printDetails()
  { System.out.println("Username: " + userName);
    System.out.println("Name: " + name);
  }
}
```

Can then be called in subclass $printDetails()$ methods as $super.printDetails()$:

```
public void printDetails()
{ System.out.println("STAFF");
  super.printDetails();
  System.out.println("Room: " + roomNumber);
  System.out.println("Ext: " + phoneNumber);
}
```

Similarly for *Student*.

Coursework 5. Hand-in: Friday 29th November, 4pm. Coursework office

1 Define a class *Person* with attributes *String name* and *int consumed* (representing the number of units of alcohol consumed in current week). Write a constructor for *Person* which sets *name* to a value provided in the constructor argument, and sets *consumed* to 0.

Define a method *public void consume*(*int units*) of this class which increments *consumed* by *units*.

Define a method *public void reset*() of the class, which resets *consumed* to 0 (eg, on Monday morning).

2 Define subclasses *Male* and *Female* of *Person*. These classes have the same attributes as *Person*, but both define a method *public int alcoholLimit*(). This method returns 14 in the case of a *Female* instance, and 21 for a *Male* instance.

3 Modify *Person* to declare *alcoholLimit* as *abstract*, and modify *consume* to give a warning message if the updated value of *consumed* is over the alcohol limit of the person, as defined by *alcoholLimit*().

Original UML Class Diagram of Person/Male/Female