Metropolitan State University, St. Paul
ICS 372 Object-Oriented Design and Implementation
Polymorphism and Dynamic Binding
Brahma Dathan

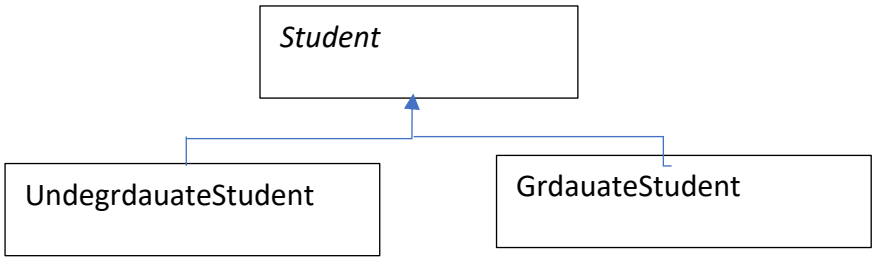
Introduction

In this short document, we look at a concept that helps you implement data structures more correctly and efficiently. The topics are not directly related to the design of data structures as we studied in the class so far; the concepts are more programming related, but, nonetheless, quite important.

We have already discussed the concept of polymorphism. In this document, we briefly review that topic and look at dynamic binding. We have used this concept many times this semester, so what this document does is really formalize the idea a bit more and emphasize it. You need to watch the video and study the full source code referred to in the document, to get a deeper understanding.

Polymorphism

Consider a university application that contains, among others, three classes that form a hierarchy as shown below. The classes are implemented in the package `dynamicbinding` in the `Data Structures` project. Please refer to the code in that package as you read through this document.



In real life, a student can be either an undergraduate student or a graduate student. The above class hierarchy represents that fact.

We can declare Java code as below.

```
Student student1 = new UndergraduateStudent("name1");
```

Or

```
Student student1 = new GraduateStudent("name2");
```

The ability of a reference to store objects of different types is called **polymorphism**. In the above, `student1` is polymorphically assigned different types of objects.

The class `StudentArrayBag` exploits this idea to store an array of `Student` objects.

```
private Student[] data;
```

Each cell of the array is a reference to a Student. The actual object referred to can be of type `UndergraduateStudent` or `GraduateStudent`. The method `add()` takes in a `Student` object as parameter and stores it in the array.

```
public void add(Student element) {
   if (numberOfStudents == data.length) {
      ensureCapacity(numberOfStudents * 2 + 1);
   }
   data[numberOfStudents] = element;
   numberOfStudents++;
}
```

The line

```
   data[numberOfStudents] = element;
```

is a polymorphic assignment.

One could use the class `StudentArrayBag` to store both `UndergraduateStudent` and `GraduateStudent` objects.

```
students.add(new UndergraduateStudent("ug student name"));
students.add(new GraduateStudent("ug student name"));
```

Some Crucial Background Concepts

There are two concepts that some readers may not be completely comfortable with. Both concepts distinguish between two situations.

Actual and Declared Types

In the statement,

```
Student student1 = new UndergraduateStudent();
```

`student1` is declared to be of type `Student`. So, we say that the **declared type** of `student1` is `Student`. However, the reference ends up pointing to an object of type `UndergraduateStudent`, so we say that when the statement is fully executed, the **actual type** of `student1` is `UndergraduateStudent`.

The declared type of a reference never changes within its scope (that is, where the reference is valid), whereas the actual type can change because of polymorphism.

Compilation Time and Execution Time

Consider the following code.

```
StudentArrayBag students = new StudentArrayBag(10);
Scanner scanner = new Scanner(System.in);
int someInput = scanner.nextInt();
if (someInput == 1) {
   students.add(new UndergraduateStudent("ug student name"));
} else {
   students.add(new GraduateStudent("g student name"));
}
```

In the first line, the code creates a bag. The program snippet then creates a `Scanner` object, so it can read input typed in through the keyboard. It then reads an integer from the keyboard and stores it in `someInput`. If the value read in is 1, an `UndergraduateStudent` object is created and added to the bag; otherwise, a `GraduateStudent` object is created and added to the bag.

If you type this code into Eclipse, the Java compiler verifies that you are following the language rules. For example, it checks if there is a semi-colon at the end of statements. As another example of a check, it also looks at the declared type of the reference `students` (which is `StudentArrayBag`) and verifies the existence of a method named `add();` it is able to confirm that both of the statements

```
   students.add(new UndergraduateStudent("ug student name"));
```

and

```
   students.add(new GraduateStudent("g student name"));
```

are legal because there is a method

```
public void add(Student element) {
```

in the `StudentArrayBag` class.

Note that the compiler restricts itself to checking the syntax and generating the byte code. *What it cannot do is figure out which statements will get executed*, let alone their order of execution. In particular, it cannot know what input value will be read, and, therefore, which of the statements (`students.add(new UndergraduateStudent("ug student name"));` or `students.add(new GraduateStudent("g student name"));`)gets executed. As

a result, it cannot know whether a `GraduateStudent` object or an `UndergraduateStudent` student objects gets inserted into the bag.

Dynamic Binding

Before reading further, watch the video that explains the structure of the application implemented in the `dynamicbinding` package.

Examine the following code.

```
for (int index = 0; index < 12; index++) {
  students.getDataAt(index).addCourse(4,
                            ((int) (Math.random() * 1000)) % 4);
}
```

You can verify that the method call `students.getDataAt()` returns a `Student` object, so the declared type of the return type is `Student`. The actual type of the object could be `UndergraduateStudent` or `GraduateStudent`. Notice that the returned object is used to call the method `addCourse()`. You can check that there is a declaration of `addCourse()` in `Student`, `UndergraduateStudent`, and `GraduateStudent`. So, a natural question to ask is which of these three methods is used for the method call. There is a simple answer for all instance methods.

At execution time, the version of the method used for execution is the one found in the class for the actual type of the object on which the method call is issued. If there is no declaration of the method in that class, the method found in the immediate superclass is used; if there is no method in that class either, its superclass is checked, and so on, until a version is found in one of the superclasses. Note that that this search will end successfully, because otherwise, the line would not have been successfully compiled.

Applying the above rule, we find that if an `UndergraduateStudent (GraduateStudent)` object is returned by the code, the `addCourse()` method of `UndergraduateStudent (GraduateStudent)` will get called.

Note that this is what we would really want. It is correct to invoke the `addCourse()` method of `GraduateStudent` if the object has the actual type of `GraduateStudent`. The above rule shows the power of dynamic binding.

Conclusions

We override methods when we think the code in the overriding methods are more appropriate than what is provided by the corresponding (overridden) method of the superclass. Dynamic binding helps us invoke the methods associated with the actual type of an object without having to specify the type to be used in the source code.