



CS 315

Software Design

Homework 5

Subtyping

Due: Oct. 15, 11:30 PM

Objectives

- Designing subtype hierarchies
- Working with Java generics

Java Generics

Generics are a feature in Java that allows the classes to be parameterized to restrict the types of variables and parameters used within the class. Without generics, one often is forced to use the type "Object" and then to use the "instanceof" operators to check the actual type of an object and casting operators to convince the compiler and runtime system to allow you to treat an object as a more specific subtype. This style of programming easily leads to runtime errors. The introduction of generics means that the same programming errors will instead result in compiler errors, meaning the developer will be certain to know about their existence.

For example, before the introduction of generics, all of the Collection types, like ArrayList, Set, List, etc. were assumed to contain any Object. The signatures of the methods in these classes looked like this:

```
public boolean add (Object o)
public Object get (int index)
```

If you wanted a collection to contain only Employee objects, you would need to be very careful when adding objects to the collection that the object was an employee. When extracting an object, you would need to "cast" the result to an Employee, like this:

```
List employees = new ArrayList();
...
Object o = employees.get(i);
if (o instanceof Employee) {
    Employee emp = (Employee) o;
    ...
}
```

In addition to the awkwardness of this code, you were also left with the quandary of what to do if you pulled an item out of a list and it happened to not be an employee.

With generics, the compiler can guarantee that only Employees will be in the list. Your code becomes much simpler:

```
List<Employee> employees = new ArrayList<Employee>();
...
Employee emp = employees.get(i);
```

Since you have declared employees to be a list of employees, it is no longer necessary to check that the object is an employee when you extract it.

The example above shows the syntax for declaring a variable or parameter:

```
List<Employee> employees
```

It also shows the syntax for constructing a new list that contains a specific type of element:

```
new ArrayList<Employee>()
```

In addition to using the generic types that Java provides, you can create your own. The syntax for doing that looks like this:

```
public class MyClass<T> {
    public void myMethod (T[] param1, List<T> param2) ...
    ...
}
```

In this example, we are declaring a generic class, MyClass. T is the name of the type parameter. Within the definition of the class, we are declaring a method. The first parameter of the method must be an array for which the type of the elements is the same as the type parameter and whose second parameter must be a List whose members are the given type parameter. Given this declaration, we could use the class like this:

```
MyClass<String> myObject = new MyClass<String>();
String[] myArray = new String[5];
List<String> myList = new List<String>();
myObject.myMethod (myArray, myList);
```

Assignment

There are three separate questions in this assignment.

1. (35 points) Modify the Sorter program given on pages 73 and 74 of the textbook to use Java's generics. The version from the textbook can be downloaded from the schedule page of the course website.

Specifically, the Comparator interface should be declared as:

```
public interface Comparator<T> {
    public int compare (T o1, T o2);
```

```
}

```

This says that `Comparator` requires a type parameter. Whatever type parameter is used then determines the types of the two parameters in the `compare` method. For example, the `IntegerComparator` class must now be written as:

```
public class IntegerComparator implements Comparator<Integer> {
    public int compare(Integer o1, Integer o2) {
        ...
    }
}

```

Here, `Integer` is the type parameter specified as "implements `Comparator<Integer>`". As a result, the `compare` method requires two `Integer` parameters.

For this question, you should:

- Modify the `compare` method of `IntegerComparator` to conform to the new signature of the `compare` method.
 - Modify `StringComparator` to implement the `Comparator` interface with `String` as the type parameter.
 - Modify the `Sorter` class to have a type parameter.
 - Modify the `Main` class to use your new `Sorter` class.
 - Define a new implementation of `Comparator` that sorts strings based on their length, so that shorter strings appear before longer strings. If 2 strings have the same length, it does not matter in which order they are sorted. Add a test of this new comparator to the `Main` class.
 - Add javadoc comments to all of the classes.
2. (25 points) Do chapter 3, question 8. Your solution should not require using the "instanceof" operator. It should also not require you to cast any objects into a subtype. For this question, I would like you to turn in a UML class diagram as well as the code that you write. Your test program should demonstrate that you can walk an array containing a mix of regular and student subscribers outputting their names and what their subscription rate is. A regular subscription should be \$20 and a student subscription should be half that amount.

Here are the original versions of the `Person` and `Student` classes that you should modify:

```
public class Person {
    private String name;

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

public class Student {

```

```

private Person me;
private double gpa;

public Student(String name) {
    me = new Person(name);
}

public String getName() {
    return me.getName();
}

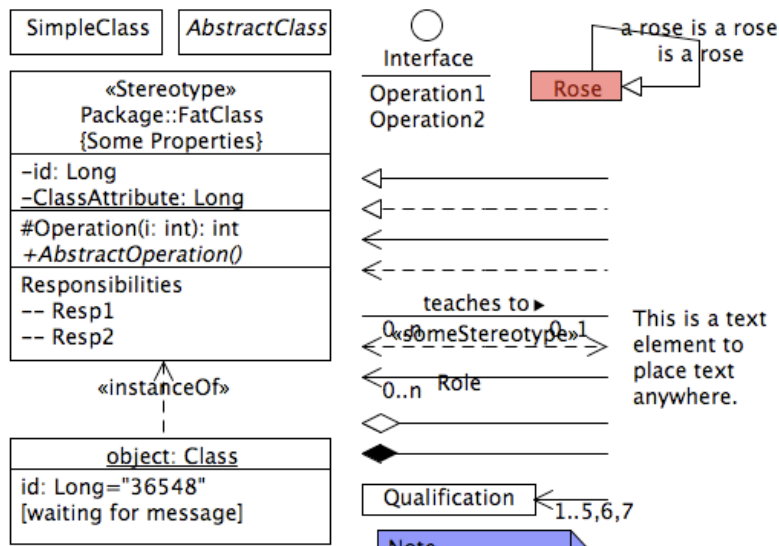
public double getGPA() {
    return gpa;
}
}
    
```

- (40 points) Do chapter 3, question 21. For this question, you should turn in a UML diagram showing the original design proposed in the question, a UML diagram showing your proposed new design, and a discussion of why your design is better than the original design.

Eclipse

You may draw your UML diagrams by hand if you like. There are a number of free tools that you can use to draw UML diagrams as well. One such tool is UMLet, which is an Eclipse plug-in. You can download it from <http://www.umlet.com/>.

The user interface to UMLet is a little odd. It will place a panel in the upper right corner of Eclipse that contains a lot of UML icons, looking like this:



Double-click on the icon that you want and a copy of it will appear in the main editing window. You can drag the icons around, but to change their content, you need to edit what appears in the panel below the icon panel. For example, if you click on the icon for a class

(the one that beings <<Stereotype>>), the panel below the icon window will contain:

Use normal text editing commands to change the contents of this panel and the contents of the UML diagram will change. For example, you can delete everything listed under responsibilities as well as the -- line above Responsibilites. You can also delete "Package::" and the "{Some Properties}" line.

Please refer to Appendix A of the text book for the proper syntax of class diagrams.

If you do not like UMLet, another simple standalone editor you can use is Violet, which is also free. You can download it from <http://sourceforge.net/projects/violet/>.

Turning in your work

Any work that you do electronically, you should turn in on Ella. If you draw your UML diagrams by hand, you may turn in paper copies in class on October 13.

```
<<Stereotype>>
Package::FatClass
{Some Properties}
--
-id: Long
--ClassAttribute: Long_
--
#Operation(i: int): int
/+AbstractOperation()/
--
Responsibilities
-- Resp1
-- Resp2
```