**chap 8 Interacting objects : Newton’s Lab**

**Objectives**

* Understand another polymorphism context: method overloading
* Understand the two usage of Java keyword “this” : one is for self-referencing, and the other is for one constructor to invoke another constructor, i.e., an overloaded constructor
* Know how to define and use constant variables in Java using keyword ***static final***
* Understand how to use Interface as superclass (parent class)
* Understand the dynamic binding and what design patterns to use for class and interface design.
* Understand the various class and interface design patterns

**Schedule:** This lesson covers Moodle folder “chap 8”. Please follow the steps below.

1, work on sec 8.1 ~ 8.9, which include scenario Newtons-Lab-1, Lab-2 and Lab-3.

2, in Lab-3, you need to understand the code to bounce the planet when it reaches the edges of the space

3, in Lab-3, you need to understand the code in class Obstacle that determines where a planet body has hit an obstacle or not.

4, after class exercise: work on textbook exercises 8.35, 8.40, 8.41 and 8.44. You need to first open the existing scenario “Newtons-Lab-3”, then start working on these exercises.

5, unzip file “**exampleOfClassInterfaceDesign.zip**”, you will see three source code folders, plus one file named “**versionComparison.docx”.**

You need to study the source code examples in these three folders, to learn the different class-interface design principles, and more specifically, you need to know which design version can accommodate dynamic binding in the application file, and which version cannot.

File “**versionComparison.docx”** illustrates the difference in the three versions of class and interface design, and presents the UML class diagram for each design.

6, the source code in this zip “**exampleOfClassInterfaceDesign.zip**” includes Interface, and the following links explain the characteristics of Interface:

<http://www.tutorialspoint.com/java/java_interfaces.htm>

<https://docs.oracle.com/javase/tutorial/java/concepts/interface.html>

We can think of Interface as a special kind of class, and its special characteristics are:

* in Interface, all methods must be abstract methods, regardless we explicitly place the keyword abstract for methods.
* in Interface, all data variables must be static final variables (i.e., must be constant variables), regardless whether we explicitly place the two keywords “static final” for variables or not.

A regular class can inherit from an Interface by using the keyword “**implements**”. The Interface and the class that inherits from the Interface also have the parent-child relationship, with the Interface being the parent or superclass, and the class being the child or subclass.

A regular class can only extend from one superclass, however, this regular class can implement multiple Interfaces.

Because the existence of the parent-child relationship, we can apply the “is-a” or “is-an” term to Interface and its child class. For example,

* in version 1, we can say that Employee “is” Payable, because Employee inherits from Payable in version 1’s design;
* but in version 2, we cannot say Employee “is” Payable anymore, because Employee class does not inherits from Payable in version 2’s design
* in both version 1 and version 2, we can say SalesPerson or HourlyWorker “is” Payable, both in both version, SalesPerson or HourlyWorker inherits from Payable. More specifically, in version 2, SalesPerson or HourlyWorker directly implements Payable Interface; and in version 1, SalesPerson or Hourly extends from Employee, and class Employee implements Payable Interface, thus the “is-a” relationship can pass from grandparent class to grandchild class.

Both java keyword “extends” and “implements” stand for inheritance relationship between superclass and subclass, i.e., the “is-a” or “is-an” relationship.

7, After a class implements an Interface, this class becomes a subclass of this Interface, and this subclass has two choices:

* Choice A: subclass overrides all the abstract methods from its superclass (the Interface), so that the subclass does not have to be declared as an abstract class.
* Choice B: subclass has at least one abstract method from its superclass (the Interface) **NOT** overridden, then the subclass has to be declared as an abstract class, and the consequence is that: the subclass cannot be instantiated because subclass is now an abstract class

From the above statement, we can see that, when it comes imposing restrictions on its subclasses, Interface is similar to abstract class. Please refer to file “**dynamicBinding-staticBinding.docx**” in Moodle folder “chap 7”, which explains the restrictions of extending from an abstract class.

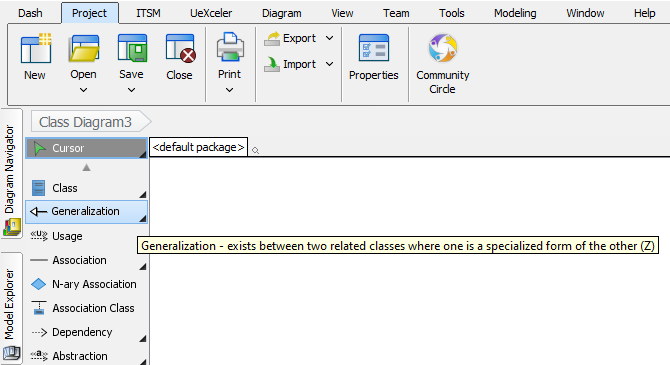
Even though Interface and abstract class have similar impacts on their subclasses, but the differences between Interface and abstract class are also obvious, as listed here:

* A subclass can extend from only one class (or abstract class), but a subclass can implement from multiple interfaces at the same time
* Inside an abstract class, you can have non-abstract methods, and non-constant variables, but in an Interface, even if you don’t have the keyword abstract for an method, or keyword “static final” for a variable, the method is still abstract in Interface, and the variable is still “static final” in Interface

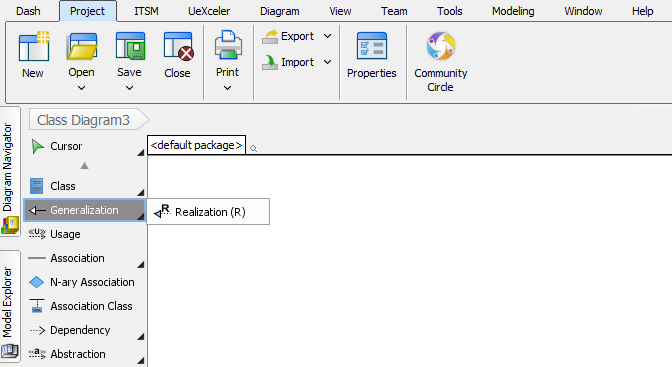
8, in UML class diagram, we distinguish the scenario where you inheritance from a class (or abstract class) from the scenario where you inheritance from an interface.

Question: where is the Generalization icon containing a triangle shape at the top with a solid dash line?

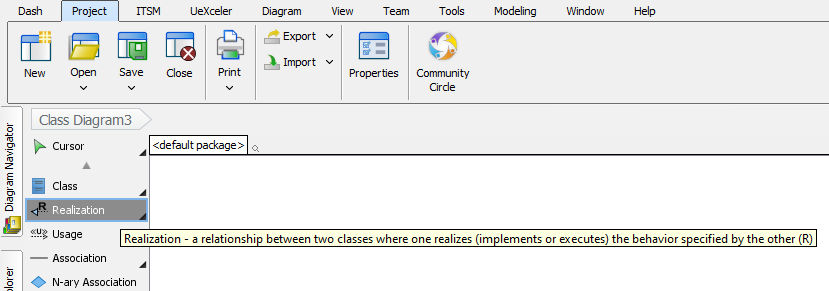
Answer: the Generalization icon is right below the Class icon in Visual Paradigm, as indicated below:



This screen show below indicates how to switch to Realization icon: by clicking the small black triangle on the lower right corner of Generalization icon, you will see the Realization icon.



After you choose Realization icon from the above picture, you are now switched to Realization icon.



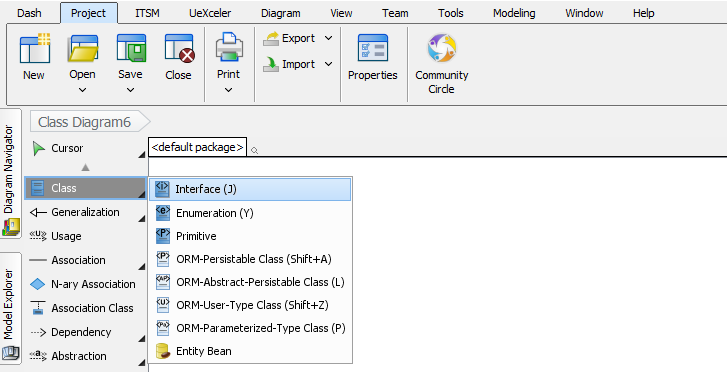
Both Generalization and Realization icons are used to indicate inheritance relationship, and the difference is:

* Generalization icon is used when a subclass inherits from a regular class or from an abstract class, so the generalization icon represents the Java keyword “***extends***”
* Realization icon is used when a subclass inherits from an interface, so the generalization icon represents the Java keyword “***implements***”

Generalization is also referred to as strong-inheritance, while Realization as weak-inheritance. And both are representing the inheritance relationship, thus we can apply the “is-a” relationship in both scenarios.

For example, we can say, SalesPerson “is-an” Employee, and we can also say, SalesPerson is Payable, or Employee is Payable, under the java source code in version 1 of “**exampleOfClassInterfaceDesign.zip**”. And you can see that, the “is-a” or “is-an” can be applied across multiple generations, such as between a grandchild class and a grandparent class or grandparent interface.

Just like you can switch between the Generalization icon and Realization icon, you can also switch between the Class icon and the Interface icon, as indicated by the picture below:



9, follow the instructions in file “**homework8.docx**”, and work on homework 8. After you finish it, you need to submit the solution zip file to its Moodle drop box. When coding your homework, please follow all the rules in file “RulesForIndentAndAlignCode.docx”.

10, keep working on the questions in file “Test2StudyGuide.docx”, and prepare for test 2. This file is available in Moodle folder “test 2 review lesson”.

11, use Eclipse, work on coding exercise 6.1 to 6.4 on polymorphism principle of OOP from this link:

<http://www.ntu.edu.sg/home/ehchua/programming/java/J3f_OOPExercises.html#show-toc>

For this link, first work on exercise 6.1 by yourself. You need to generate the source code based on the given UML class diagram for abstract class Shape, then the subclasses class Circle, Rectangle, and Square. When it comes to protected data in a class, you just change the requirement to private data. Then you need to create an application class called TestShapes, which has the main method, and in the main method, you include the piece of code given in the link. If any code does not compile in the main method, you need to explain why it does not compile, and then you need to correct the error to make it compile. Then run the application file and explain the output.

Then you can work on exercise 6.2 to 6.4, which may include interface into the class inheritance structure. Each exercise has its own implementation requirements, and please follow these requirements carefully.

Another link addressing polymorphism is: <http://www.ntu.edu.sg/home/ehchua/programming/java/J3b_OOPInheritancePolymorphism.html#zz-5> , and there are more explanations about abstract class and interface and polymorphism in OOP in section 5 of this link.

12, next lesson, we will be working on Moodle folder “chap 9”, please study it in advance.