Dynamic Binding and Static Binding

After you unzip file “***dynamicBinding-staticBinding.zip***”, you will see this WORD document, plus two java source code folders: “*exampleOfDynamicBinding*” and “*exampleOfArrayList*-*StaticBinding*”. You need to study this WORD document carefully, along with the source code files in the above two folders.

Before we talk about dynamic binding or static binding, we need to know the definitions of these items first:

* abstract method
* abstract class

, and their official definitions are in this link: <https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html>

I summarize their characteristics as below:

For abstract method:

* It has the keyword abstract in its method header
* Is method header has to be terminated by semi-colon “;”, and it cannot have a pair of curly brackets { }. Even it is a pair of curly brackets with no statement inside, it is still wrong.

For abstract class:

* It has the keyword abstract in the class definition line
* An abstract class cannot be instantiated, that means, you cannot have a statement like this:

AbstractClassName objName = new AbstractClassName ( parameters of constructors…);

* It may or may not have any abstract method.
* If an abstract class has any abstract method, then its subclass has two choices:
  + Choice A: subclass overrides all the abstract methods from its superclass, so that the subclass does not have to be declared as abstract class
  + Choice B: subclass has at least one abstract method from its superclass **NOT** overridden, then the subclass has to be declared as abstract class, and the consequence is that: the subclass cannot be instantiated because subclass is now an abstract class.

In summary, if an abstract method appears in superclass, then all its subclasses have the obligation to override this abstract method, therefore, a unified API is enforced into its subclasses. Otherwise, if any of the abstract methods from superclass is not overridden, then the subclass itself has to be declared as abstract, and the obligation to override the abstract method is now passed to the third generation, i.e., the subclass of subclass.

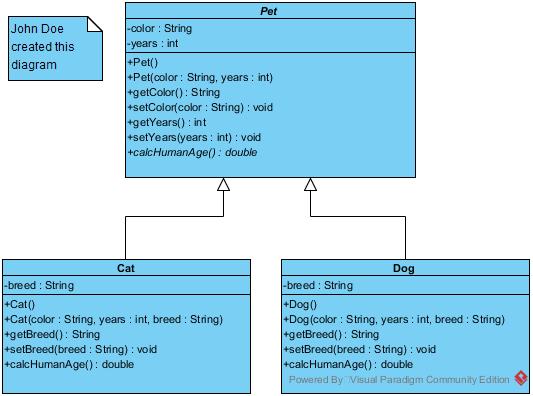
In folder *“exampleOfDynamicBinding”*, we have three classes with inheritance relationship, plus one other class “TestPet.java” as the application class that makes use of the three classes. The UML class diagram of these three classes is shown in the next page.

As you can see, there is an abstract method in superclass Pet (class Pet itself is an abstract class):

* *public abstract double calHumanAge()*

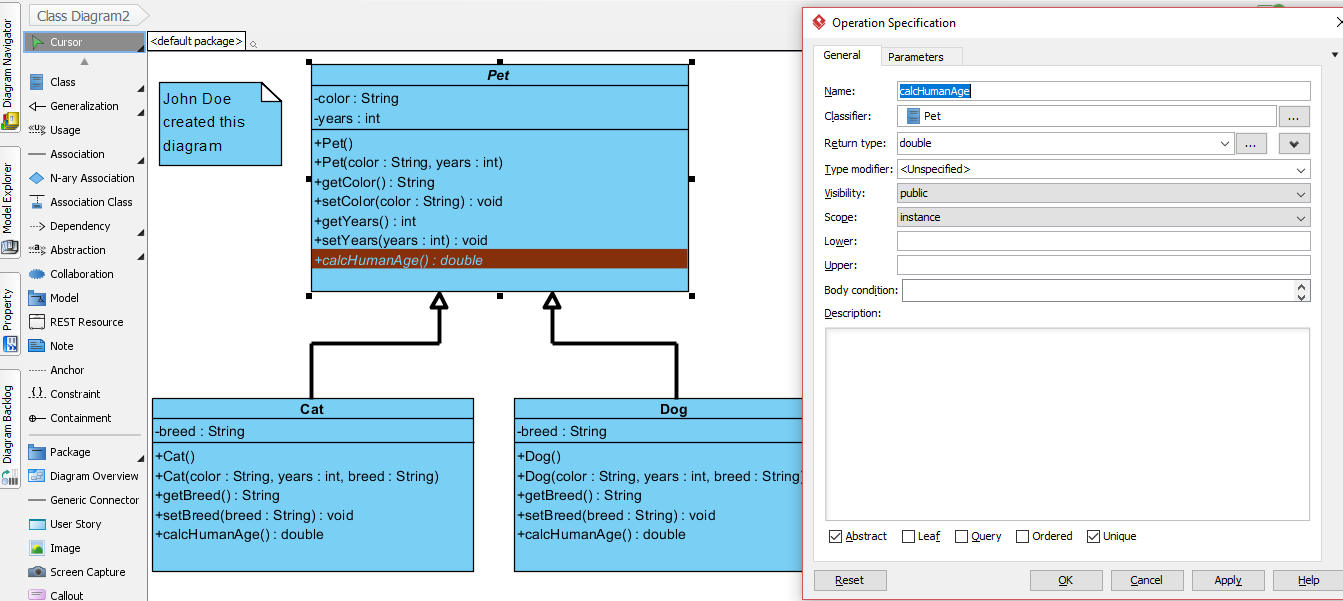
Pet has two subclasses, class Cat and class Dog, and both subclasses override the abstract method, in other words, they have concrete implementation of the abstract method. Therefore, the Cat class and the Dog class don’t have to be defined as abstract class anymore, and they are both non-abstract concrete classes.

Notice that, the abstract method ***calcHumanAge()*** in superclass Pet is ***italic*** in UML class diagram, which indicates that it is abstract. In two subclasses Cat and Dog, the method **calcHumanAge()** is no longer ***italic***, which means that it is a concrete implementations of the abstract method from the superclass, in other words, it overrides the abstract method from the superclass.



Also notice that class ***Pet*** itself is an abstract class, therefore, the class name ***Pet*** is ***italic*** in the UML class diagram.

To indicate a method or class as abstract in UML class diagram, you just need to right click the class name or method name, then choose menu item “Operation Specification”, and then click the “Abstract” checkbox on the lower left corner of the pop up window, as indicated below:



The table bellows summarizes the terminologies we used for dynamic binding and static binding:

|  |  |  |  |
| --- | --- | --- | --- |
| Synonyms for static binding -------- | Early binding | Pre binding | Compile-time binding |
| Synonyms for dynamic binding -------- | Late binding | Post binding | Runtime binding |

In summary, when the binding of the source code with the actual method body happens in compile-time, it is static binding; and when the binding happens in run time, it is dynamic binding.

Please look at folder “*exampleOfDynamicBinding*”, file “***TestPet.java***”, line 21, and notice that the binding of the calcHumanAge() method with the correct method body, either the 7 \* getYears() version of calculating the cat’s equivalent age, or the 9 \* getYears() version of calculating the dog’s equivalent age, only happens in runtime. Because only in runtime, after line 15 or line 18 has been executed, can the JVM/JRE know in line 21 that whether the pet is a cat or a dog.

Please look at folder “*exampleOfArrayList-StaticBinding*”, file “***ArrayListDemo.java***”, and notice that line 26 and 27 need to be commented out, because ArrayList object ***aList*** is static-binding to its element type **BankAccount**, and ArrayList object ***bList*** is static-binding to its element type **Integer**, and object ***aList*** and object ***bList*** cannot swap their binding element types. If you uncomment line 26 and 27, then these two lines won’t pass the Java compiler. So basically, the definition in line 9 for generic class object ***aList*** is in contradiction with the usage in line 27 for this generic class object ***aList***, and the definition in line 10 for generic class object ***bList*** is in contradiction with the usage in line 26 for this generic class object ***bList***.

Therefore, we say that the generic class type is static binding, because in compiling time, the compiler will check the class type we plug in when using a generic class (which is line 17, 18, 26, 27 in ***ArrayListDemo.java***), to see whether it matches the class type in the definition of generic class (which is line 9, 10 in ***ArrayListDemo.java*** ).

Generic class type, such as ArrayList, always has a pair of angle brackets < > to surround the element type that is static binding to the class, such as :

ArrayList<BankAccount> aList = new ArrayList<BankAccount>();

ArrayList<Integer> bList = new ArrayList<Integer>();

At the end of this lesson, let’s review the four principles of OOP, as summarized in step 14 in file “chap1-schedule.docx”.

**Four principles** of OOP are **data abstraction, encapsulation, inheritance,** and **polymorphism**, as shown below:

* **data abstraction**
* **encapsulation**
* **inheritance**
* **polymorphism** with the following context:
* **method overloading**
* **method overriding**
* **parent reference refers to child object**
* **dynamic binding**
* **static binding** (the other name is generic programming)

As you can see, we have covered all four OOP principles, including all the context of polymorphism.

How can you tell whether you have understood these OOP principles or not?

You can try this: if you can explain each OOP principle / context in your own words, together with sample java source code, then you can be very confident that you have a good understanding of OOP principles.