Day 11: Polymorphism

1 Polymorphism

Polymorphism is the ability of an entity to show diferent shapes or forms. In biology, polymorphic animals can exist in a variety of different forms, e.g. male and female animals, or different blood types. In materials science, polymorphic materials can exist in a variety of different crystaline configurations (e.g. calcite and aragonite).

In computer science, polymorphism appears at many levels. In Programming in Java, we are mainly interested in two types of polymorphism: polymorphic classes and polymorphic methods. The former is usually referred to as *generic types*, while the latter is usually called *method overloading*.

2 Generic types/classes

A generic type or generic class is a complex type that can be combined with other different types to produce a new type. The most common application of generic types is for "container" classes like lists, stacks, and queues.

When we were creating our own dynamic classes in previous weeks, they were usually attached to a type: lists of strings, stacks of integers, etc.

```
// Part of a String List
public class StringNode {
    private String value;
    private StringNode next;
    // ...
}
```

```
// Part of an IntegerList
public class IntegerNode {
    private Integer value;
    private IntegerNode next;
    // ...
}
```

This seems a bit like code repetition. A *generic class*, on the other hand, could work in the same way with more than one type, saving the programmer from the hassle of having different but very similar types (e.g. stacks) for each different type of data they want to use (i.e. string stacks, integer stacks, patient stacks, etc).

2.1 Using generic classes

A generic class is very similar to a normal class. The only difference —but it is a crucial one— is that it uses one or more *type parameters* to indicate that it needs another type to be fully declared. Type parameters are specified in low quotes ("<", ">"). Let's see an example of a generic interface, a stack (JavaDoc comments removed for brevity):

```
public interface Stack<T> {
    void push(T object);
    T pop();
}
```

The type parameter `T` represents a type to be determined when the interface `Stack<T>` is used. For example, we could declare a `Stack<String>` or a `Stack<Integer>` (see below).

If we wanted to implement this interface using arrays, it could look like this (comments removed for brevity):

```
public class ArrayStack<T> implements Stack<T> {
    private T[] contents;
    private int headIndex;

    public void push(T newElement) {
        contents[headIndex] = newElement;
        headIndex++;
    }
    public T pop() {
        if (headIndex == 0) {
            return null;
        }
        headIndex--;
        T result = contents[headIndex];
        contents[headIndex] = null;
        return result;
    }
    // ... constructor and other methods would be here ...
}
```

As you can see, there are two main differences between a normal class and a generic class (and their methods):

- The name of the class is extended to include the type, by using a type parameter in low quotes. Capital `T` (for "type") is a common name for type parameters. By convention type parameters are usually capital single letters (T for "type", K for "key", V for "value", etc), but any valid identifier can be used.
- The variable type can be used in every place where a type (simple or complex) should be: parameter types, return types, and for declaring new variables.

How are generic interface and classes used? Look at this code:

```
// ...
Stack<Integer> myNumberStack = new ArrayStack<Integer>();
Stack<String> myStringStack = new ArrayStack<String>();
myNumberStack.push(1);
myStringStack.push("Test string!");
// ...
```

As you can see, the same generic interface and generic class can be used to handle different specific interfaces/classes, i.e. stacks of integers and strings in this case. Generic types must be based on complex types (classes), not simple types. Boxed types are useful if you need a generic type based on one of the primitive (i.e. simple types). This is why the example uses a `Stack<Integer>`; if you try to create a `Stack<int>`, Java will complain.

2.2 Restricting generic types

There are situations in which you want to create a generic type that can only be combined with some specific types. For example, you may want to use a list of `Person` in your program, but you do not want lists of string or hospitals appearing in your code from other modules.

Restricting the type is easy and can be done in more than one way. For now, we will look at just one way of doing it by means of interfaces. Look at this example:

```
public interface PersonList<P extends Person> {
    void push(P object);
    P pop();
}
```

In this case, we are using P instead of T for the type parameter to make it clearer that it must be a Person, not just any type. Note that the keyword to retrict the type parameter is `extends`, not `implements`. The compiler knows that any type that we specify for `PersonList<P>` must be a class that extends/implements the interface `Person`. Therefore, we will be able to create a `PersonList<KidPerson>` but not a `PersonList<Integer>`. If we try to do the latter, Java will complain.

2.3 Type wildcards

Generic classes can also be used in a generic way (no pun intended) by using *type wildcards*. For example, we may want to write a generic method that takes a generic-class parameter inside a non-generic class, as in this example, where a `School` class has a method that always returns the first element of a list, regardless of it being a list of students, of teachers, or of homework assignments:

```
public class School {
    // ...
    public T returnFirstElement(List<T> stack) {
        return stack.get(0);
    }
    public int returnElementCount(List<T> stack) {
        return stack.size();
    }
    // ... more methods here
}
```

If you try to compile this method, the compiler will complain because it does not know what the type parameter T is. There are three ways of solving this problem. First, we could make School a generic class:

```
public class School<T> {
    // ...
```

This is a bad idea in this case because classes should not be made generic unless there is a good reason for it. Generics add an additional level of complexity in the code that must be balanced with some clear benefit —as it is the case with container classes such as lists, stacks, trees, maps, etc). Another, better, possibility is to generify only the method `returnFirstElement()`:

```
public class School {
    // ...
    public <T> T returnFirstElement(List<T> stack) {
        return stack.get(0);
    }
    // ... more methods here
}
```

This is slightly better because it keeps the generics more contained, i.e. restricted to the method.

If the method returns a generified type, restricting generics to the method is the best we can do. However, if the return type is not generic, we can use an even simpler notation using type wildcards:

```
public class School {
    // ...
    public int returnElementCount(List<?> stack) {
        return stack.size();
    }
    // ... more methods here
}
```

A type wildcard (i.e. a question mark) means that the generic type can be specified with any other type. Wildcards can be restricted if necessary:

```
public class School {
    // ...
    public int returnElementCount(List<? extends Person> stack) {
        return stack.size();
    }
    // ... more methods here
}
```

In this case, the method will accept a list of any type as long as the type is a `Person`. The third way is the best approach in this case because it is the simplest, the one that hides the most information without losing any functionality.

2.4 Conclusion

Generic classes are one of the main types of polymorphism present in Java. They enable programmers to combine one class with several other classes, by using type parameters.

Generic classes are, therefore, a way of reusing code and/or avoiding code repetition (remember the DRY principle: Don't Repeat Yourself). There is no need to create a list of integers, a list of strings, and a list of books. It is better to create a generic list that can be combined with any type, and then combine it with integers, string, books, or any other type as needed. The drawbacks of generic classes is that they are slightly more verbose (i.e. the programmer needs to write more) and that they can become very confusing if abused, e.g. when generic classes of generic classes of generic classes are employed.

A full description of generics in Java, how they are implemented internally, and the full list of possibilities and caveats, is beyond the scope of this course. As a rule of thumb, remember these three simple rules:

- You cannot create new objects of generic types. Code like `new T()` or `new T[]` will generate a compile error.
- You cannot create local variables of generic types. Code like `T newNumber = new Integer(1)` will generate a compile error.

And remember these two pieces of advise:

- Use generic *sparingly* in your Java code, and always with good reason.
- Do not abuse generics creating generics types of generics types (your brain will thank you in the long term).

3 Method overloading

Another very common form of polymorphism is *method overloading*. This is just a fancy name to say that a method can receive different parameters. We have seen several examples already, such as the method `substring` of class `String`. This method can take either one or two `int` parameters; depending on the number of arguments, the behaviour is different.

```
// ...
String str = "This is some text";
System.out.println(str.subString(8));    // prints "some text"
System.out.println(str.subString(8,12)); // prints "some"
// ...
```

A method in Java is defined by its name and the types and positions of its parameters; therefore, strictly speaking, `substring(int)` is a different method from `substring(int,int)` and it is not a problem if both are present in class `String`. The following methods, however, do have the same *signature* and there can be only one version in each class:

```
// Same method! Cannot be in same class.
String myMethod(int first, int second, String message);
String myMethod(int start, int end, String txt);
```

Remember that the names of the parameters are not important: only their types and positions. Note that the return type is not part of the signature either, so we cannot have two methods with the same name and parameters but different return types.

```
// Same method! Cannot be in same class.
int myMethod(int first, int second, String message);
void myMethod(int first, int second, String message);
```

Method overloading without repeating code One thing that a programmer must bear in mind when method overloading is used is not to repeat code. For example, the following overloaded method repeats code unnecesarily:

```
public void printAverage(double d1, double d2) {
     double result = 0.5 * (d1 + d2);
     System.out.println("The average is: " + result);
}
public void printAverage(double d1, double d2, String msg) {
     double result = 0.5 * (d1 + d2);
     System.out.println(msg + result);
}
```

There is repeated code among both versions of the method. In this trivial example this is just one line, but in a more serious program this could be some non-trivial algorithm that is repeated, and we know that code repetition must be avoided.

Avoiding this kind of code repetition is easy by making the more specific version of the method call the more general version, as in the code below:

```
public void printAverage(double d1, double d2) {
     printAverage(d1, d2, "The average is: ");
}
public void printAverage(double d1, double d2, String msg) {
     double result = 0.5 * (d1 + d2);
     System.out.println(msg + result);
}
```

This is how we can have multiple polymorphic versions of the same method without any repetition of code. Remember: Don't Repeat Yourself.

4 Upcasting and downcasting

There is a third type of polymorphism (of sorts) in Java that is simpler but also very important. We have already seen how it works, although we have not used its proper name until now. We have been using it every time we have declared a variable by using an interface or a superclass:

```
List patientList;
patientList = new ArrayPatientList();
```

We know that it is good practice to work with interface names for our data types, rather than to use the class names directly. Using interfaces allows programmers to abstract themselves from all the implementation details of the class and work with just its public behaviour, i.e. the methods defined on the interface. This is called *upcasting*, and it is similar to the casting of simple types we

have already seen; you just need to know that, by convention, interfaces are imagined to be "up" and specific classes are imagined to be "down" (see Figure 1), so casting from class to interface is up-casting. (The "origin" in computer science is almost always up: the root of a tree, the interface of a class, etc).

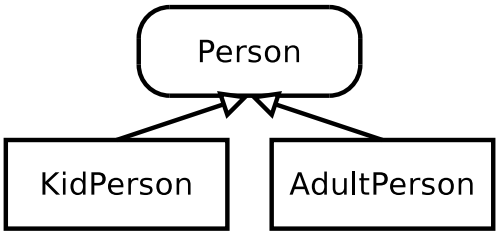

Figure 1: Interface Person is implemented by classes KidPerson and AdultPerson. By convention, the interface is represented "up" and the classes are represented "down".

Another useful form of upcasting is when a programmer uses an interface as a type for a generic class:

```
Stack<Person> myPersonStack = new ArrayStack<Person>();
```

On the left-hand side, we have a generic interface (`Stack<T>`) that is made specific by using another interface (`Person`). The variable `myPersonStack` will point to a stack that will only store objects of classes that implement the `Person` interface (e.g. `KidPerson`, `AdultPerson`, etc).

Upcasting is a way of "moving up to know less", i.e. to intentionally forget details about the implementation so that we can work with a simpler, more generic data type. Bear in mind that a real program is really complex, and therefore reducing complexity is usually in the programmer's benefit; that is why upcasting is so common in Java programs.

In some rare occasions, programmers need to do the opposite, i.e. move down from the general to the specific: this is called *downcasting*. The syntax in this case is very similar to the casting of simple types. Imagine that we had a list of animals, but we knew (somehow) that all those animals were dogs, and we wanted to make them bark. As `bark()` is a method of `Dog` but not of `Animal` we would need to downcast them:

```
public void makeThemBark(List<Animal> animals) {
    for (int i = 0; i < animals.size(); i++) {
        Animal nextAnimal = animals.get(i);
        Dog dog = (Dog) nextAnimal;
        dog.bark();
    }
}
```

Upcasting is an operation that detects errors at compile time: if the code compiles, it will always run. This is not true of downcasting, which can fail at *runtime*, even if it compiled successfully. In the example above, the method `makeThemBark(...)` may receive a list of `Animal` where not all the elements are really objects of class `Dog` (maybe there is a `Cat`). When the code tries to downcast a class into something that it is not, Java will throw a `ClassCastException`:

```
java.lang.ClassCastException: Cat cannot be cast to Dog
```