Polymorphism – Abstract Data Types

COSC 1P03 – Lecture 05 (Spring 2024)

Maysara Al Jumaily

amaysara@brocku.ca

Brock University

Tuesday May 28, 2024

Total slides: 17

Lecture Outline

What is an Interface

- An interface is a way to achieve abstraction by specifying what instance variables and methods classes must implement.
- It is the blueprint/outline of a class.
- It *only* contains `public` instance variables and/or `public` method header, no code! These methods achieve abstraction as they don't contain code.
- An interface can be completely empty with no instance variables nor method header.
- A simple example of an interface (note, we use the `interface` keyword):

```
public interface Calculator {
  public int zero = 0;//one instance variable (could have more)

  //Three methods to implement
  public int add(int x, int y);
  public int subtract(int x, int y);
  public int multiply(int x, int y);

}
```

Implementing an Interface I

- To "satisfy" the requirements of an interface, you need to create a class that `implements` it (*i.e.*, link an interface to the class), as such:

```
public class MyCalculator implements Calculator {
  //some instance variables if needed
  public MyCalculator(){
    //some initialization
    //...
    //we could print out the variable in the interface
    System.out.println(Calculator.zero);
  }

  public int add(int x, int y){
    return x + y;
  }
  public int subtract(int x, int y){
    return x - y;
  }
  public int multiply(int x, int y){
    return x * y;
  }
}
```

Implementing an Interface II

- The meaning of implementing is having the class to **literally** have the **exact** same method header. By that, we mean:
  - Implement **every** single method header found in the interface (forgetting one or more method(s) gives a compilation error).
  - The access modifier of the method **must** be `public` (since an interface only allows for public instance variables and method header).
  - The return type of the method **must** be the same type that is found in the interface.
  - The method name **must** match the name found in the interface (yes, case-sensitive).
  - The number of parameters **must** match the ones found in the interface.
  - Parameters must have the same type and the order found in the interface. For example, if the interface has:

    ```
    public void print(int x, double y, String z);
    ```

    then, the only way to implement that method is by writing:

    ```
    public void print(int x, double y, String z){...}
    ```

    The line below will *not* work because the order doesn't match:

    ```
    public void print(double x, int y, String z){...}
    ```

Implementing an Interface III

- Suppose that our `MyCalculator` class implements the `Calculator` interface and we have the three methods `add`, `subtract` and `multiply` to be implemented successfully.
  - We know that having less than the number of methods we want to implement gives an error as we didn't satisfy all of the requirement of the interface.
  - We also know that if our class implements exact number of methods in the interface, it all works correctly.
  - The important question is that... what if we have more than the required methods, would that work?
    - Remember, our task is to satisfy the requirements given in the interface.
    - Hence we *don't* violate the requirement if we have more methods, so it is allowed!

Interfaces seem extra work for no reason I

- So far, it seems that we are doing a lot of extra work for no reason.
- Since we know that we need to have the `add`, `subtract` and `multiply` methods, why do we just have them in the class and not even have an interface.
- That way, we satisfy the requirements needed without an interface.
- Well... code-wise, yes, we can satisfy the desired output without using an interface.
- However, we will lose on a very important concept (mentioned in The advantage and usefulness of Interfaces V).

Interfaces seem extra work for no reason II

- We might be referring to something like (a class *without* an interface, which mean without the '`implements Calculator`' part):

```
public class MyCalculator {
  //some instance variables if needed
  public MyCalculator(){//some initialization
  }
  public int add(int x, int y){
    return x + y;
  }
  public int subtract(int x, int y){
    return x - y;
  }
  public int multiply(int x, int y){
    return x * y;
  }
}
```

- This will work. When we want to create an instance of `MyCalculator` class, we will write: `MyCalculator c = new MyCalculator();`
- Nothing special there. It is important to note that the type left to the equal sign (`MyCalculator`) matches the type right of equal sign.

The advantage and usefulness of Interfaces I

- Suppose that we will have the following `Animal` interface:

```
public interface Animal {
  public void sayName();//prints name of animal: cat, dog, bird, etc.
  public boolean canFly();
}
```

- Also, assume that we have a `Cat` class that implements the `Animal` interface as such:

```
public class Cat implements Animal {
  public Cat(){
    //some initializations if needed
  }
  public void sayName(){
    System.out.println("Cat");
  }
  public boolean canFly(){
    return false;//a cat doesn't fly
  }
}
```

The advantage and usefulness of Interfaces II

- We want to run the code, how to create an instance of our object?
- The nïave and not as correct way to do it is by writing:

```
Cat c = new Cat();//left type equals the right type
```

- This still works but we are missing the point of interfaces.
- We cannot do the following either:

```
Animal c = new Animal();//left type equals the right type
```

because `Animal` is an interface and it doesn't contain any code, we cannot run anything!

- The correct way is to create the type to be an `Animal` and initialize it to a `Cat` object, as such:

```
Animal c = new Cat();//left type doesn't equal right type
```

- This example by itself might not help. So, the next slide shows a complete but minimal example of the power of interfaces. We will have the same `Animal` interface along with the `Cat` class and also create a `Dog` class (which implements `Animal`) and a `Bird` class (which also implements `Animal`).

The advantage and usefulness of Interfaces III

- The `Dog` class :

```
public class Dog implements Animal {
  public Dog(){ ... }
  public void sayName(){ System.out.println("Dog"); }
  public boolean canFly(){ return false;}
}
```

- The `Bird` class :

```
public class Bird implements Animal {
  public Bird(){ ... }
  public void sayName(){ System.out.println("Bird"); }
  public boolean canFly(){ return true;}
}
```

The advantage and usefulness of Interfaces IV

- To summary:
- The `Animal` interface is:

```
public interface Animal {
  public void sayName();//prints name of animal: cat, dog, bird, etc.
  public boolean canFly();
}
```

- The `Cat` class implements the `Animal` interface:
  - writes `"Cat"` for `sayName()`.
  - returns `false` for `canFly()`.
- The `Dog` class implements the `Animal` interface:
  - writes `"Dog"` for `sayName()`.
  - returns `false` for `canFly()`.
- The `Bird` class implements the `Animal` interface:
  - writes `"Bird"` for `sayName()`.
  - returns `true` for `canFly()`.

The advantage and usefulness of Interfaces V

- How can we create an instance of our `Animal` type?
- Remember, doing `Animal a = new Animal();` doesn't work because an interface doesn't have code to execute.
- We need to keep the type an `Animal` but when creating an instance of a class, we have three options.
- All the lines below will work:

```
Animal c = new Cat();
Animal d = new Dog();
Animal b = new Bird();
```

- Furthermore, we can even change the instance type of the same object:

```
Animal a = new Cat();
a = new Dog();
a = new Bird();
```

- In case we have more time, we could go over creating a generic `Shape` example.

Access Modifiers

- Access modifiers are the `public`/`private` keywords we use to associate with instance variables and methods
- There are four classifications: `private`, `public` and `protected` (`protected` is used in methods only), and default (writing none of the three options)
- `private` means the instance variable/method is accessible only within the class
- `public` means the instance variable/method is accessible everywhere to any classes that are within the package or outside of the package
- default (*i.e.*, not writing any of `public`, `private` or `protected`) means the instance variable/method is accessible within the package only, and that other packages cannot access it
- `protected`, is similar to default but also allows subclasses to access the instance variable/method (this deals with inheritance, COSC 2P03)

Creating Exceptions

- There are times where we need to throw a custom error when a specific scenario is encountered
- It will not be `NullPointerException`, not `IndexOutOfBoundsException`, but some custom error
- For example, let us say we want to create a custom exception when trying to remove from an empty linked list and call it `EmptyLinkedListException`
- We have to create a new exception that `extends` a class that it created by the Java developer called `RuntimeException`
- We will create a class named `EmptyLinkedListException` which `extends` the `RuntimeException` class:

```
public class EmptyLinkedListException extends RuntimeException{
  //For COSC 1P03, don't put anything else :)
}
```

- We will use it in the next slide

Throwing an Exception

- To use the exception from the previous slide, we need to create it by using the keyword `throw`
- Suppose that we have the following code that checks if the linked list is empty and if yes, then crash the program:

```
private void remove(int value){
  if(head == null){
    throw new EmptyLinkedListException(); // crashes the program!!
  }else{
    //perform the actual code of removing an element
  }
}
```

- The code above will crash the program
- While it is good to crash, it is not practical to have the app crash on the user
- Instead, we need to handle the error once it is created, by using the `try`/`catch` block

Catching an Exception

- To catch an exception, we will use the `try`/`catch` block, which is given as:

```
try{
  //risky code that is possible to crash
}catch (ErrorType e){
  //handling the error appropriately instead of crashing
}
```

- The `ErrorType` is the name of the exception class (in our case `EmptyLinkedListException`)
- It could be `NullPointerException`, or `IndexOutOfBoundsException` or any other error that is possible to encountered
- The name `e` is the variable name associated with the error
- This is similar to passing some parameter which needs a type and variable name
- The idea is that the developers *creating* the package or library need to throw errors, and the programmers *using* those libraries catch it