Object-oriented programming and data-structures

CORNELL UNIVERSITY FOUNDED A.D. 1865

CS/ENGRD 2110
SUMMER 2018

Lecture 4: OO Principles - Polymorphism

http://courses.cs.cornell.edu/cs2110/2018su

Lecture 3 Recap

- Good design principles.
  - Modularity
  - Encapsulation
  - Inheritance

- Access modifiers, **extends**, constructor chaining, etc.

Lecture 4

- Abstraction
- Polymorphism
- Multiple Inheritance Problems
- Interfaces
- Parametrised Types

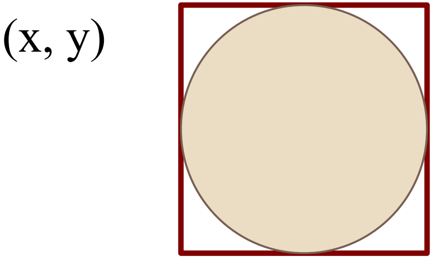
Inheritance - Recap

- Inheritance allows types to be specialised
  - Minimise code re-use
  - Allows multiple specialised types (ex: instructor, student) to be used everywhere the base class can be used.
- But has some shortcomings …

A geometry detour

- Assume that want to write a geometry program that can manipulate the **area** and **perimeter** of 2D **shapes.**
  - Want to define **circle, rectangle** and **triangle**

Position of a rectangle in the plane is given by its upper-left corner. Calculate perimeter by 2*(width + height), area by width * height



Position of a circle in the plane is given by the upper-left corner of its bounding box. Perimeter calculated by 2*Π*radius, area by Π*radius^2

A geometry detour - Inheritance?

Create a class shape that defines area() and perim() functions, and have every subclass extend Shape and override those methods.

What’s wrong here?

Shape
x
y
area()
perim()

Rectangle
area()
perim()
width
height

Triangle
area()
perim()
base
height

Circle
area()
perim()
radius

Inheritance - Recap

- Inheritance allows types to be specialised
    - Minimise code re-use
    - Allows multiple specialised types (ex: instructor, student) to be used everywhere the base class can be used.

- Inheritance can
    - force a **family of derived classes** to implement specific functionality
    - But there isn't really a convenient **default behaviour** for the base class.

Abstraction

- Program specification mandates any class that is a **Shape** should implement **area()** and **perimeter()**
  - But area() and perimeter() of a Shape doesn't really make sense

- Instead, want to **force** all **Shapes** to implement their own **area()** and **perimeter()**

- Shape is an **abstract type** with certain desired functionality
  - Square, Circle, etc. are **concrete instantiations** of that type

Abstract classes to the rescue

- Most OOP languages support a notion of **abstract classes**
  - Abstract classes can contain **method stubs** (methods without a body)
  - Abstract classes cannot be **instantiated**
    - Why?

- Java uses keyword **abstract**

  -
    ```
    class abstract Shape {
            int x ; int y;
            int getXPosition() { return x;}
            abstract int area();
    }
    ```

Syntax:
If a method has keyword abstract in its declaration, use a semicolon instead of a method body

Multiple Inheritance

- Examples so far suggest that a class can inherit from a single base class

- Sometimes, want to inherit from multiple base classes
  - Meet **the graduate student**
    - Can be both a **Student** and an **Instructor**
  - What can we do?

Diamond Inheritance Problem

- Multiple inheritance introduces **the diamond problem**
- **Definition**: Ambiguity that arises when a class inherits from two classes that define and implement the same method.

```
class Instructor  extends Person {
        int salary;
        int getSalary();
        void dance() { System.out.println(“Let’s boogie”);}

}
```

```
class Student  extends Person {
        int gpa;
        int getGpa();
        void dance() { System.out.println(“Let’s cha-cha”);}
}
```

GraduateStudent inherits from both Instructor and Student. Should she boogie or cha-cha?

Interfaces to the rescue

- Java mandates that every class can inherit from at most one **class** (possibly abstract)
  - Instead, it introduce “special classes” that can do multiple inheritance: **interfaces**

- **Definition** Interfaces are special classes that have
  - no state (cannot define any fields)
  - all methods are abstract

- Interfaces define **functionality only**, a **contract** that any concrete types must satisfy

Interfaces to the rescue

- Java uses **interface** keyword to define an interface
- Classes must **implement** an interface

```
public interface A {
	public int myMethod();
}
```

```
public class B implements A  {
	public int myMethod();
}
```

Revisiting the Graduate Student

- A graduate student can **teach**
  - Implements a Teaching interface with method getSalary()
- A graduate student can **study**
  - Implements a Study interface with a method getGPA();
- A graduate student is still a **Person** (hopefully)
  - Extends class Person, inherit fields name, DoB

```
class GraduateStudent extends
Person implements Teaching,
Study {
        ...
}
```

Interfaces vs Abstract Classes

- Not going to lie, they are similar. Hard to determine which one to use at times
  - We'll see next lecture two examples of Java Interfaces
  - Rule of thumb: when it doubt, start with an interface

- View interfaces as:
  - what something can do/defines an abstract data type/contract to fullfill
  - force high-level of abstraction in code
- View abstract classes as:
  - represents something
  - allows sharing common code between subclasses

- What should Shape be? Interface or abstract class?

Manipulating derived types

- Recall: inheritance allows us to use derived types everywhere we want to use a base type.

`public int sumAreas(Shape[] allShapes)`

- Consider a method:
  - Want to calculate the sum of the areas of all the shapes in the drawing
  - But area() is an abstract method and all shapes implement different area methods. What can we do?

First attempt - Casting

- Explicitly try casting each individual shape to the appropriate type
  - instanceof keyword
- Downsides:
  - Cumbersome to write, error-prone
  - Every time add a new Shape, have to modify that function

```
public int sumAreas(Shape[] allShapes) {
    int sum  = 0;
    int nbShapes = allShapes.length;
    for (int i = 0 ; i < nbShapes ; i++ {
        Shape s = shape[i];
        if (s instanceof Circle) {
            Circle c = (Circle) s;
            sum+= c.area();
        } else if (s instanceof Triangle) {
            Triangle t = (Triangle) s;
        } else { … }
      }
    }
```

Polymorphism to the rescue

- Definition: a language's ability to process objects of various types and classes through a single, uniform interface

- Java polymorphism calls the appropriate method for the type of the object that is referred to in each variable rather than the method that is defined by the variable's type
  - Shape s = new Circle(); s.area() will call the circle area method.

- Polymorphism
  - *separates the interface and implementation*
  - allows the programmer to *program at the interface* only

Second attempt

- Magic of polymorphism

- Only need to worry about the spec of Shapes (they all implement an area() method). Not about any specifics of the Shape
  - Better modularity
  - Less buggy

```
public int sumAreas(Shape[] allShapes) {
        int sum  = 0;
        int nbShapes = allShapes.length;
        for (int i = 0 ; i < nbShapes ; i++ {
                Shape s = shape[i];
                sum+= s.area();
        }
        return sum;
}
```

Static vs Dynamic Polymorphism

- Java uses **dynamic polymorphism**
  - Run the method in the child
  - Must be down at runtime since that is when you know the child's type.
- Alternative is **static polymorphism**
  - Decide at compile-time.
  - Since don't know what true type will be, just run the method in the parent type.

- Dynamic polymorphism much more practical, but has a performance overhead
  - Java only does dynamic
  - C++ offers developers the choice

Principles of OO Recap

- May all seem similar
  - Modularity
  - Encapsulation
  - Abstraction
  - Polymorphism

- All sides of the same coin: enable clean, easy to reason about with minimal bugs, where each object has well-defined functionality and exposes only the minimal information necessary to other components of the system.

References in JavaHyperText

abstraction

abstract class

interface

implements

extends

polymorphism

subtyping

abstract data type