Jin L.C. Guo

M2 (a) - Types and Polymorphism

Image Source: https://upload.wikimedia.org/wikipedia/commons/2/2b/Cepaea_nemoralis_active_pair_on_tree_trunk.jpg

Recap of last module

- Programming mechanisms:
  - Scope and Visibility
- Concepts and Principles:
  - Information Hiding, Encapsulation, Escaping Reference, Immutability
- Design Techniques:
  - Object Diagrams
- Patterns and Antipatterns:
  - Primitive Obsession 👎

Objective of this lecture

- Concepts and Principles:

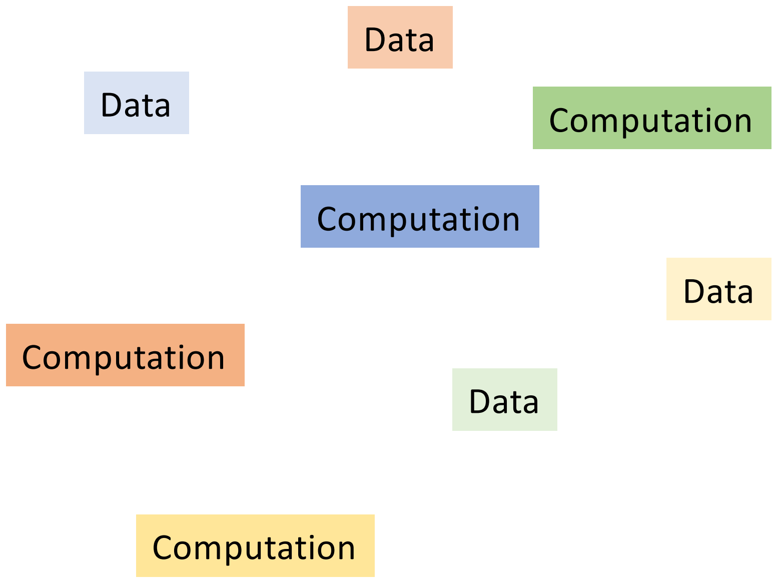
Class's interface, Separation of concerns

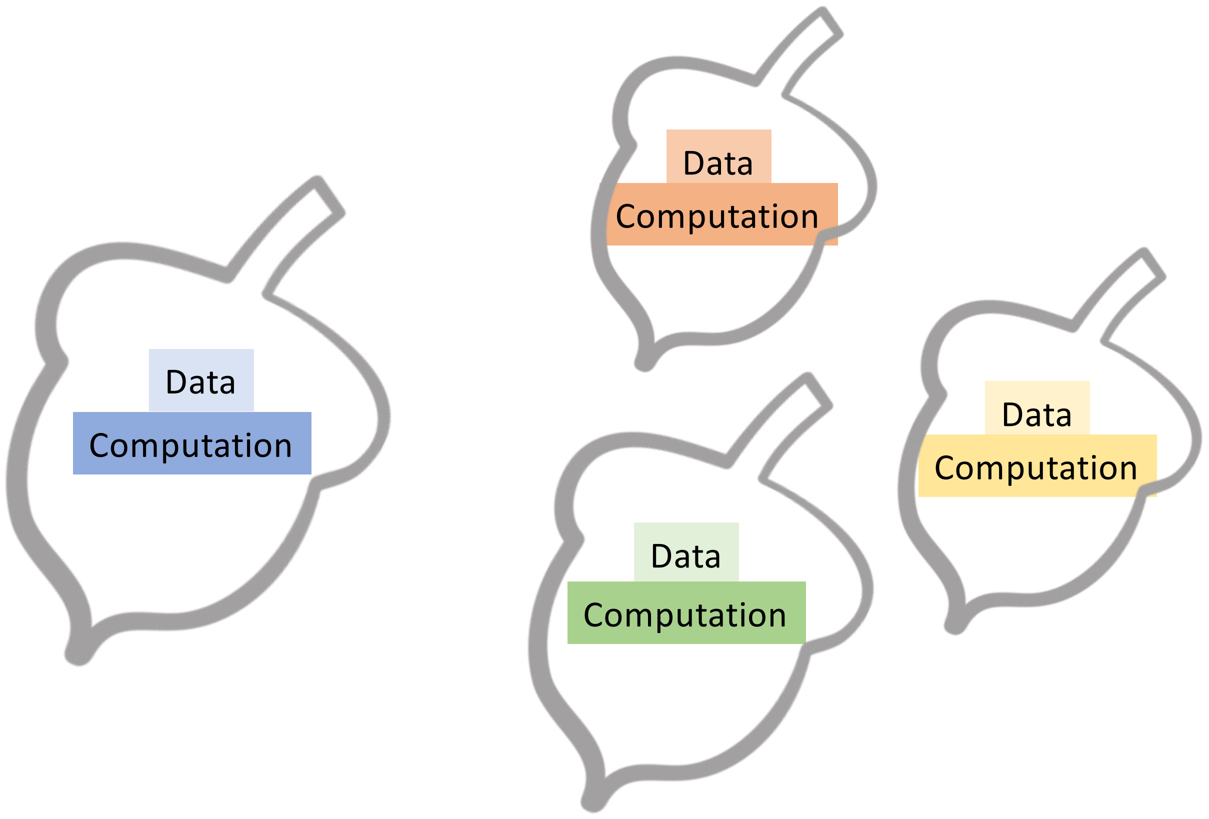
- Programming mechanism:

Java Interface type, Subtype polymorphism

- Design techniques:

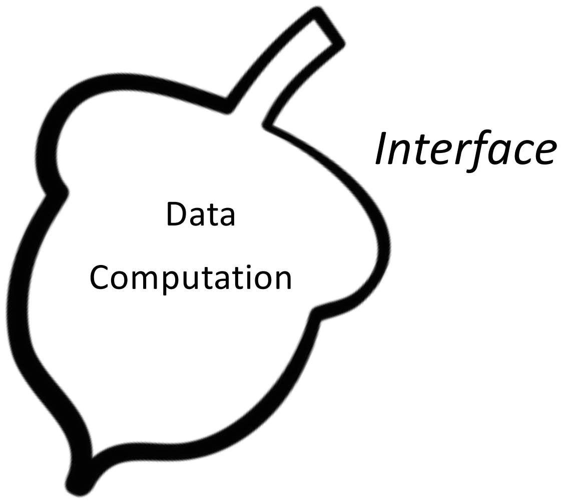
Interface-based behavior specification, UML Class Diagrams


Data
Data
Computation
Computation
Data
Computation
Data
Computation



Functions are achieved through Object Interaction

Object Interaction



Supply the service through public interface

Activity 1:

Which ones should be public?

- Thank about the design of a Class **`StudentGradeRecord`**, which provides the basic functions related to the students' grade. What kind of methods should this class provide?

Add the grade

Iterate through grades for an existing course

Update the grade

View the existing grade for a certain course

Validate the grade

Print out the grade to a file

Calculate the GPA

Open a file

Write to a file

What should the public interface specify?

• Requires	What needs be true in order to call this the method?
• Modifies	When this method is called, is the state of any object going to be changed?
• Effects	What will happen if this method is called?

Add the grade

Objective of this lecture

- Concepts and Principles:

Class's interface, Separation of concerns

- Programming mechanism:

Java Interface type, Subtype polymorphism

- Design techniques:

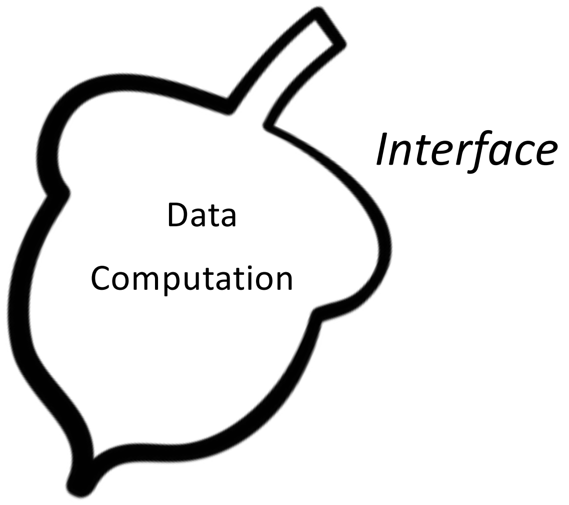
Interface-based behavior specification, UML Class Diagrams

Java Interface Type

- Specification of related methods
- Reference to invoke those methods
- No implementation yet (except default and static methods)

```java
public interface Student {
    /*
    * @return The unique id associated with student
    */
    String getID();
    /*
    * @return The first name of the student.
    */
    String getFirstName();
    /*
    * @return The Last name of the student.
    */
    String getLastName();

}
```



Subtype Relationship

```
public class Undergrad implements Student
```

```
Student s1 = new Undergrad();

String id = s1.getID();
```

1. `Undergrad` need to provide implementation of methods in `Student`
2. Objects of `Undergrad` can be referred using variables of type `Student`.

Undergrad is-a **Student** (subtype relation)

Why do we need this?

```java
public class Undergrad implements Student

public class Graduate implements Student

public class NonDegreeStudent implements Student

public class VisitingStudent implements Student
```

Polymorphic **Student**

Extensibility

Loosely coupling

Program to the interface

```java
public boolean attendSeminar(Student pStudent)
{
    if(registeredStudents.size()<=cap) {
        registeredStudents.add(pStudent.getID());
        return true;

    }
    return false;
}
```

Polymorphism

- Many + Forms
- In programming languages, it’s the ability to present the same interface for different underlying types.

Image source: Griffith Ecology Lab (https://griffithecology.com/)

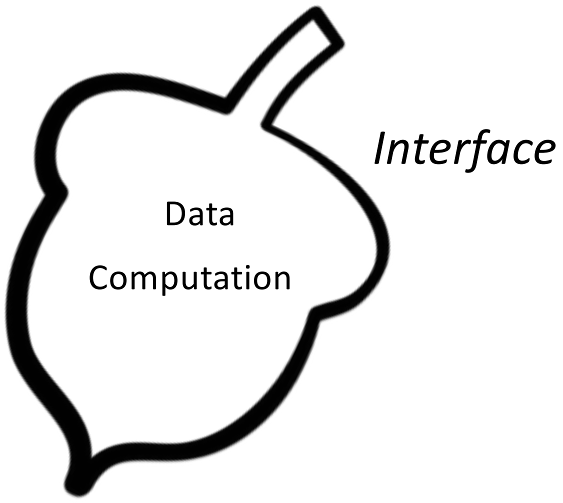
Comparison between Subtype and Subclass

- Subtype is about substitution:
  - B is a subtype of A means that if whenever the context requires an element of type A it can accept an element of type B.

- Subclass (one type of subtype) is about inheritance:
  - B is a subclass of A means that B can reuse unchanged fields and methods from A.
  - Extra dependencies between A and B
  - More in Later Modules (about Inheritance)

```
public class SpecialDeck extends Deck {

        … …
}
```



What computation should be specified in the interface?

Look for **Orthogonality**

Operation on a Deck

Draw

Add

Remove

Shuffle

Sort

Search

Max/min

Compare
two elements to get
their order or equality

Information leaking
a design knowledge is reflected in many modules

Java `Comparable<T>` Interface

- This interface imposes a total ordering on the objects of each class that implements it.

```
public interface Comparable<T>
{
        int compareTo(T o);
}
```

Generics: mechanism that takes type as parameter

Specification of `Comparable<T>`

- Compares this object with the specified object for order.
- Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
- Also properties of implementor needs to ensure, for example:

  (x.compareTo(y)>0 && y.compareTo(z)>0) implies x.compareTo(z)>0

Client

```
    if(object1.compareTo(object2) >0) /*...*/
```

Implements `Comparable<T>`

```
public interface Comparable<T>
{
        int compareTo(T o);
}
```

```
Collections.sort(aCards);// aCards is a List<Card> instance
```

```
public class Card implements Comparable<Card>
{

 … …


    @Override
    public int compareTo(Card pCard)
    {
    … …   return aRank.compareTo(pCard.aRank);
    }
}
```

Objective of this lecture

- Concepts and Principles:

Class's interface, Separation of concerns

- Programming mechanism:

Java Interface type, Subtype polymorphism

- Design techniques:

Interface-based behavior specification, UML Class Diagrams