Introduction to UML and Class Diagrams

Engineering 5895
Faculty of Engineering & Applied Science
Memorial University of Newfoundland

UML

- Unified Modelling Language (UML)
- UML is a graphical modelling Language
  - **graphical** --- UML documents are diagrams
  - **modelling** --- UML is for describing systems
  - **systems** --- may be software systems or domains (e.g. business systems), etc.
- It is **semi-formal**
  - The UML definition tries to give a reasonably well defined meaning to each construct

Three Ways of Using UML

- UML as sketch
  - Used to sketch out **some** aspects of the system
  - Create diagrams only for important classes and interactions
- UML as blueprint
  - Complete design for the whole system
  - Interfaces for all subsystems specified (but not implementation!)
- UML as programming language
  - Diagrams compiled directly to executable code!
  - Neat idea, but not yet mainstream
- We will utilize UML as sketch in this course

Classes

- Classes are specifications for objects
- Parts of a class:
  - Name
  - Set of *attributes* (*aka data members or fields*)
  - Set of *operations*
    - Constructors: initialize the object state
    - Accessors: report on the object state
    - Mutators: alter the object state
    - Destructors: clean up (not used in Java)

C++ Representation of a Class

Attributes

Operations

```
class Point {
private:
    double x, y;
public:
    /* Constructor. */
    Point(double x, double y) {
        this->x = x;
        this->y = y;
    };
    double getX() { return x; };
    void setX(double inX) { x = inX; };
    // ...
};
```


Java Representation of a Class

Attributes

Operations

```
public class Point {
    private double x, y;

    /* Constructor. */
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() { return x; }
    public void setX(double inX) { x = inX; }
    // ...
}
```


A Student Class in Java

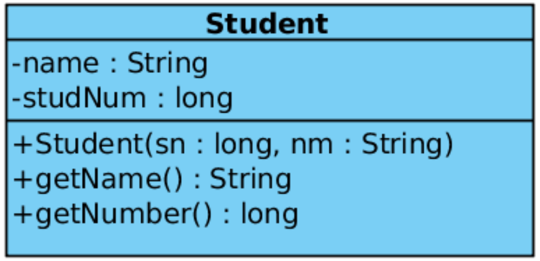
Name

Attributes

Operations

```
class Student {
    private long studNum ;
    private String name ;
    public Student( long sn, String nm ) {
        studNum = sn; name = nm; }
    public String getName() { return name; }
    public long getNumber() { return studNum; }
}
```


UML Representation of a Class



- private

+ public

UML syntax: +/- name : type

Classes in UML

UML can be used for many purposes.

- In *software design* UML classes usually correspond to classes in the code.
- But in *domain analysis* UML classes are typically classes of real objects (e.g. real students) rather than their software representations.

Usage of (Software) Classes in Java

A class `C` can be used in 3 ways:

- **Instantiation**. You can use `C` to create new objects.
  - Example: `new C()`
- **Extension**. You can use `C` as the basis for implementing other classes
  - Example: `class D extends C { … }`
- **Type**. You can use `C` as a type
  - Examples: `C func( C p ) { C q ;… }`

Relationships Between Classes

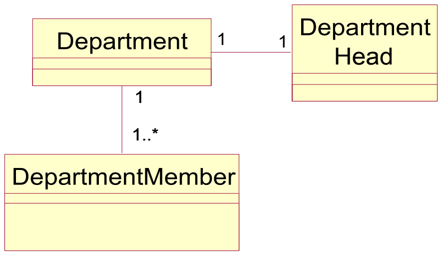
- Association
- Aggregation
- Composition
- Dependence
- Generalization

Association Relationships

- Association is a general purpose relationship between classes.
- Associations are typically named.
- Associations are often implemented with pointers (C++) or reference variables (Java)

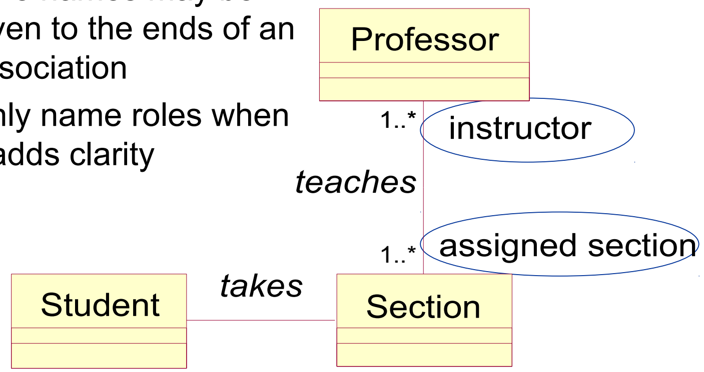
Multiplicity Constraints

- Each Department is associated with one DepartmentHead and at least one DepartmentMember
- Each DepartmentHead and DepartmentMember is associated with one Department
- No constraint means multiplicity is unspecified



Role names

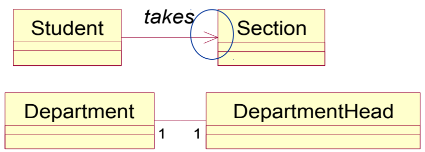
- Role names may be given to the ends of an association
- Only name roles when it adds clarity



Navigability

- An arrow-head indicates the direction of navigability.
- E.g. Given a student object, we can easily find all Sections the student is taking.

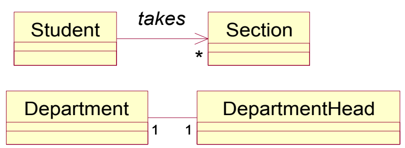
⚠ No arrow-head: means navigability in both directions.



Implementing Navigable Associations

Usually implemented with data members

```
class Student {
  private List<Section> sections; … }
class Department {
  private DepartmentHead deptHead;   … }
```



Implementing Associations Indirectly

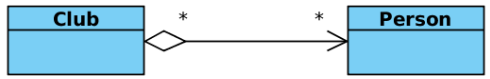
- An association between objects might also be stored outside of the objects

```
class Department {
  private static Map<Department,DepartmentHead>
    heads = new<Department,DepartmentHead>
    HashMap();

  DepartmentHead getHead() {
    return heads.get(this);
  }
  …
}
```

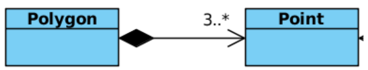

Aggregation

- Aggregation is a special case of association.
- It is used when there is a “whole-part” relationship between objects.
- Denoted with an unfilled diamond at the “whole” end
- eg. A Club is an aggregation of Persons (the members of the club)



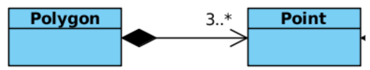
Composition

- Composition is a special case of aggregation
- Composition is appropriate when
  - each part is a part of one whole
  - the lifetime of the whole and the part are the same
- Denoted by a solid diamond at the “whole” end
- eg.
  - A Polygon is composed of 3 or more Points

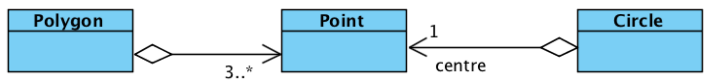


Composition vs. Aggregation

- The difference between composition and aggregation is lifetime
- For example, if whenever the points that compose it are destroyed, the polygon is destroyed (and vice versa) then we have composition

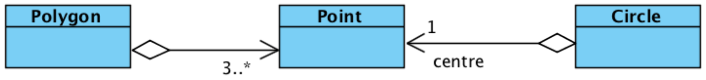


- But maybe this is not what what we want. If we allow the points to exist independently of the polygon, then we can also use them to define other shapes



Note: Class Diagrams Show Class Relationships, Not Object Relationships

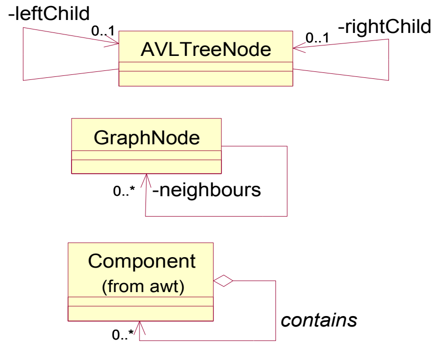
- Consider again this example:



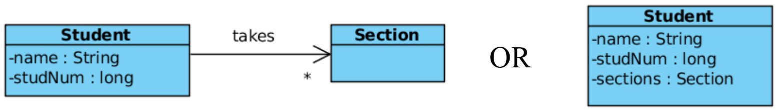
- We're not saying that the same points (i.e. instances of Point) are necessarily shared by Polygons and Circles, but they could be

Recursive associations

- Associations may relate a class to itself.
- The objects of the class may or may not be associated with themselves.
- (For example, the left and right children of a node would not be that node. But a GraphNode object might be its own neighbour.)



Associations vs. attributes



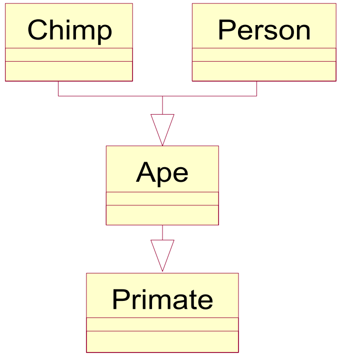
- Both are usually implemented by variables within the class
  - Fields (Java), data members (C++).
- Use association for references that point to classes or interfaces.
  - Or use aggregation or composition if appropriate
- Use attributes for primitive types such as int, boolean, char

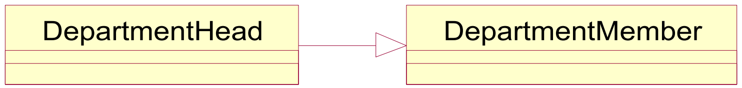
Degrees of belonging

- Attribute. Lifetime of attribute equals life time of object that contains it.
- Aggregation. Whole-part relationship, but parts could be parts of several wholes, or could migrate from one container to another.
- Composition. Lifetime of the part equals or is, by design, nested within the lifetime of the whole.
- Association. Relationship is not part/whole.

Generalization/Specialization

- Represents “is-a-kind-of” relationships.
- E.g. every Chimp is also an Ape.
- In OO implementation it represents class inheritance: Inheritance of interface and of implementation too.



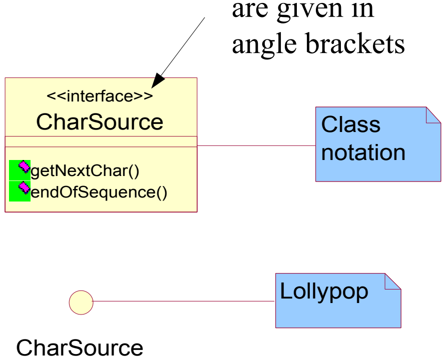


Pausing here to introduce Inheritance, Abstract Classes and Methods, and Interfaces in Java

Interfaces

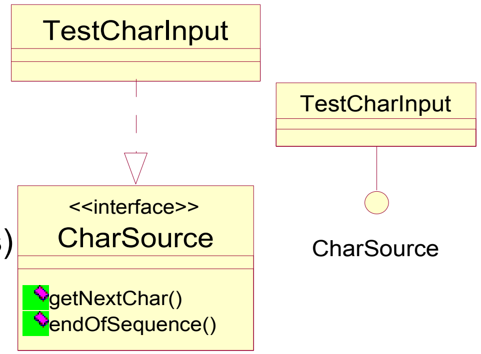
Stereotypes are given in angle brackets

- Interfaces are classes that have no associated implementation.
- I.e.
  - no attributes,
  - no implementations for any operations
- In UML use either stereotype to indicate an interface, or “lollypop”



Realization

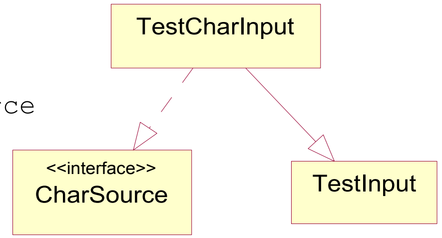
- Classes “specialize” classes, but “realize” interfaces. Similar concept, similar notation. (Note dashes)
- Choice of notations. Diagrams at right are equivalent.



Generalization/Specialization and Realization in Java

UML terminology	**Java terminology**
C specializes D	C extends D
C realizes D	C implements D

```
class TestCharInput
   extends TestInput
   implements CharSource
{
   …
}
```

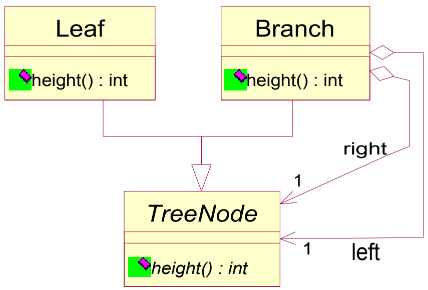


Abstract operations

- An operation O is "abstract" in class C if it does not have an implementation in class C.
- The implementation of the operation will be filled in in specializations of C.
-
```
abstract class TreeNode {
  abstract int height() ; … }
class Leaf extends TreeNode {
  int height() { return 1 ; } … }
class Branch extends TreeNode {
  int height(){return 1 + Math.max( l.height(),
                                     r.height() ; } … }
```


Abstract in Visual Paradigm (VP)

- In VP classes are made abstract with a checkbox in the specification.
- Likewise for operations (class must be abstract first).
- Italics indicate abstractness



Abstract and Concrete classes

- Classes that have abstract operations can not be instantiated --- since this would mean that there is no implementation associated with one of the object's operations
- Classes that can not be instantiated are called ***abstract classes***.
- Classes that can be are called ***concrete***
- In UML use the <<abstract>> stereotype for abstract classes and operations.
  - Alternatively: The name of the abstract class or operation is in italics.

Dependence

Dependence is the weakest form of relationship

A class C depends on class D if the implementation or interface of C even mentions D

For example if C has an operation that has a

- parameter
- local variable
- return type

of type D

Dependence

- Dependence relations are important to note because unneeded dependence makes components...
  - harder to reuse in another context
  - harder to isolate for testing
  - harder to write/understand/maintain, as the depended on classes must also be understood
- It is better to depend on an interface than on a class.
- More on this later...