CSC 335 – Classes, Interfaces, and their Modeling II

Dr. Jonathan Misurda
jmisurda@cs.arizona.edu

Arrays violate "no public fields" rule

```
int[] arr = new int[100];
for(int i=0; i<arr.length; i++) {
	System.out.println(arr[i]);
}
```

- Why `arr.length` and not `arr.length()`?
  - Maybe it is a performance issue
    - Calling functions probably takes more time than a simple field access
  - This is not our primary concern
    - Performance only matters if it isn't acceptable later on, not now

`final` fields

- We can prevent someone from modifying a field by declaring it final
- This is easily confused with the notion of constant
  - A field that is final cannot be changed, but if it is a reference, the object instance can still be altered

```
final Student s = new Student();
s.setName("Bob");     //This is fine
s = null;             // Compiler error
```

Primitive Constants

- Math.PI
  - double 3.141592653589793
- Math.E
  - double 2.718281828459045
- All-caps naming convention signals named constants
- Can initialize as part of declaration or construction, never assign again
- Probably also made `static`

`static`

- A static field or method belongs to the class rather than to any one instance
- We can use this to create named constants, utility methods, or configuration to be shared amongst all instances that are eventually created
- Solves the "bootstrapping" problem of Java
  - How do we call main if it is an instance method (non-static) if we don't have any code that creates an instance yet

Inheritance

```
class Student {
    int age;
    String name;
    float gpa;
}
```

```
class Teacher {
    int age;
    String name;
    …
}
```

```
class Staff {
    int age;
    String name;
    …
}
```

- We have three classes with a duplication of fields (and likely methods)
- This is because they are all related by specialization
  - Students, Teachers, and Staff are all people

class Person

```
class Person {
	private int age;
	private String name;
	…
}
```

Composition

```
class Student {
	private Person myself = new Person();
	private float gpa;
	…
}
```

- We could use the "HasA" composition to encapsulate the fields that we share with a person into a sub-object of our class Student

Modeling Interactions

- We can model the interactions of classes via UML
- At the core, we model this with a line connecting the classes involved in some sort of a relationship
- When we have well-defined, specific relationships, we can choose various types of lines, arrows, and labels to augment our understanding

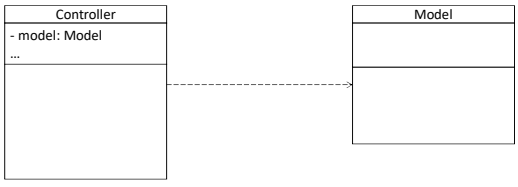
Relationships

- There are three kinds of relationships in UML
  - 1) Dependency
  - 2) Association
  - 3) Generalization
- Understanding the semantics (meaning) of these relationships is more important than simply remembering the kinds of lines that UML uses

1) Dependency: A "Uses" Relationship

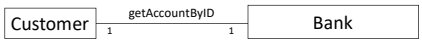
- Dependencies
  - occur when one object depends on another
  - if you change one object's interface, you need to change the dependent object
  - arrow points from dependent to needed objects
  - The line is dotted and ends as an arrow

Model/Controller



2) Association: Structural Relationship

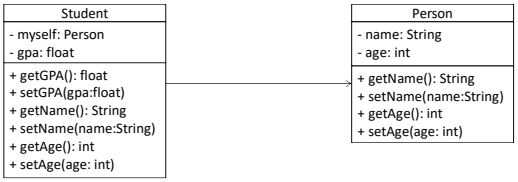
- Association
  - A relationship between classes that allows one object instance to cause another to perform an action on its behalf.
  - Specifies that objects of one kind are connected to objects of another and does not represent behavior.
  - Can label associations with a verb phrase which describes the relationship between concepts



Causation

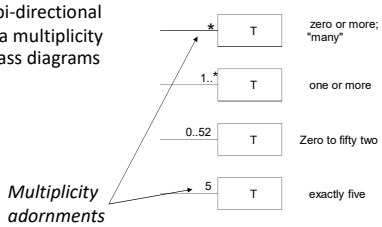
- The verb we label this relationship with maps onto our OOPLs notions of classes being able to perform actions
- To get an object to do such an action, we:
  - Send it a message
  - Invoke a method, or
  - Call a member function
- When we implement this relationship, we often must be holding a reference to one or both objects involved.

Student/Person



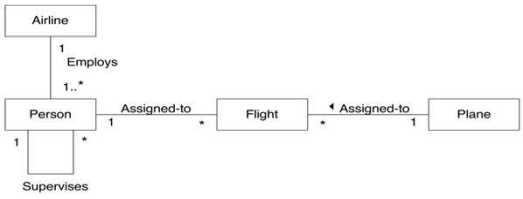
Notation and Multiplicity Adornments

- UML Associations
  - a line between two concepts and a name
  - they are bi-directional
  - can have a multiplicity
  - exist in class diagrams



Association Names

- Read this Type-VerbPhrase-Type
- Not shown here: Attributes and Methods
- This just shows associations between objects



3) Generalization

- Generalization is a relationship in which one model element (the child) is based on another model element (the parent)
- There are two types of generalization in Java
  1. Inheritance
  2. Implementing an interface

Inheritance

- Our Student "HasA" relationship to Person doesn't quite make sense.
- A Student "IsA" Person
  - We can model "IsA" via **inheritance**
- Inheritance allows us to make more specific an existing general class
  - We **extend** the more general class to be something more
  - This extension is always additive, we are never subtracting something from what we inherit

Derivation

- We call the more general class a **base class**
- We call the more specific class that extends a base class the **derived class**
- The relationship can be expressed depending on the perspective:
  - The base class is a **superclass** of the derived class
  - The derived class is a **subclass** of the base class

Basic Inheritance in Java

- We use the `extends` keyword in our class declaration to indicate that we wish to subclass another
  - The general format is:

```
class derivedClass extends baseClass { … }
```

- Every class in Java extends another
  - If not specified, you extend `java.lang.Object`
  - Thus, transitively, every class extends `Object`

Person Base Class

```
class Person {
    private int age;
    private String name;
    …
}


class Student extends Person {
    private float gpa;
    …
}
```

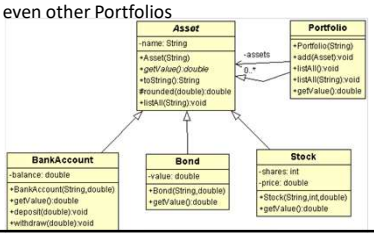
Usage

```
Student bob = new Student();
bob.setAge(20);
bob.setName("Bob");
bob.setGPA(3.1);
```

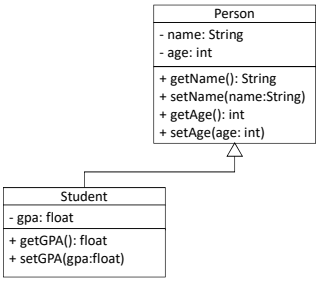
- The student instance can now do everything a student can do *plus* everything a person can do

Inheritance

- Inheritance is modeled in UML as a solid line with a hollow arrowhead that points from the child model element to the parent model element
  - Model a Portfolio which can hold BankAccounts, Bonds, Stocks, and even other Portfolios



Student/Person



Inheritance Hierarchies

- With OOP being a "world view" we might recognize that we could go further to include even more generic or specific classes
  - A Person IsA Mammal
  - A Mammal IsA Vertebrate
  - A Vertebrate IsAn Animal
  - …
- Where do we stop?

Inheritance Hierarchies

- Sometimes it makes sense to build many tiers of classes
  - This allows us to deduplicate code that might be shared among multiple subclasses
- But we're modelling the world, not reconstructing it
  - We don't need to go down to the subatomic particle level, we just need to capture things around the level we are trying to implement
- If all we want is to keep track of students and employees, we can probably stop at Person