# CSC 335 – Classes, Interfaces, and their Modeling

Dr. Jonathan Misurda
jmisurda@cs.arizona.edu

## Objects in Java

- We can build, use, and define new and existing objects in Java
- A class is a template/blueprint/cookie cutter
  - We define how an object works (implementation)
  - We define how an object is interacted with (interface)
- All code in Java must be contained within a class

## Defining a Class

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

Allows for composition of data into an aggregate form – "record type"

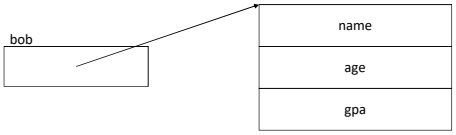
## Composition: the "HasA" Relationship

- String itself is a class from the Java Class Library
  - A Library is a collection of code we didn't have to write ourselves
- Classes allow us to create new types
- We can use those types in the definition of other classes – composition

## Create instances

```
Student bob = new Student();
```

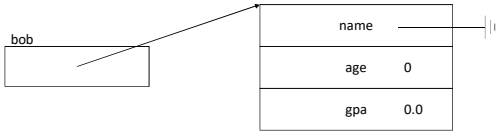
- new returns a reference to the memory allocated for the fields of the class




## Create instances

```
Student bob = new Student();
```

- new returns a reference to the memory allocated for the fields of the class



## Universal Modeling Language

- We just saw the merit in diagrammatically showing our objects
- Unified Modeling Language (UML) comes from Rumbaugh, Booch, and Jacobson (the three amigos) who combined efforts to standardize on one **modeling language**
- "UML is a language for
  - visualizing
  - specifying
  - constructing
  - documenting

  the artifacts of a software intensive system."

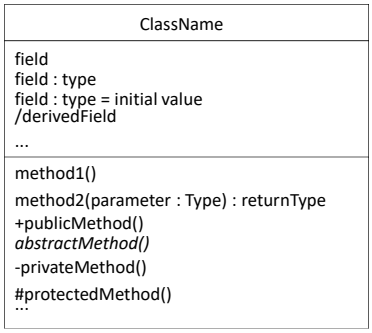
## Types of UML Diagrams

- **Class Diagram**
  - Demonstrates the relationships between classes in object-oriented software
  - Good for visualizing how classes interact
- **Package Diagram**
  - Demonstrates how the classes group into packages
  - Good for visualizing how classes are organized
- **Sequence Diagram**
  - Demonstrates the flow of control through the system, usually along the major branch of execution.
  - Good for visualizing when objects interact
- **State Diagram**
  - Displays the sequences of states that an object of an interaction goes though during its life

## Even More Diagram Types

- **Use Case Diagrams**
  - display the *relationship* among *actors* and *use cases*.
- **Collaboration Diagrams**
  - another form of interaction diagram (as are sequence diagrams); they display *interactions between objects*, with numbers to show the sequence of messages (method calls).
- **Activity Diagrams**
  - a form of state diagram in which most of the states are *action states*.
- **Component Diagrams**
  - show relationships among *source code* components, *binary* components, and *executable components*.
- **Deployment Diagrams**
  - display the configuration of *run-time processing elements*.

## Class Diagram




## Fields

- A UML field has up to five components:

1. **Visibility**: The scope of the field
2. **Type**: a base type or other class
3. **Name**: an identifier for the field
4. **Multiplicity**: If absent, then the field has exactly one value. Otherwise, [*lower* .. *upper*], where *lower* and *upper* are integers or named constants, or simply `[ ]`
5. **Initial value**: If present, preceded by =

## Full Field Specification

```
-greetings[1..3]: Greeting = Greeting.DEFAULT
```

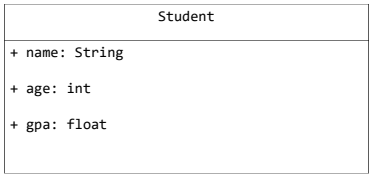
Visibility (private) — Name (greetings) — Multiplicity (may have 1 through 3 greetings) — Type (Greeting class) — Initial Value (A static default)

## Java vs UML

- In UML, when something is left out, it is intentionally ambiguous, and that is considered acceptable. In Java, there is either an implicit default or an error.

| Visibility | UML notation | Java syntax | Meaning |
|---|---|---|---|
| **public** | + | public | Visible to class |
| **private** | - | private | Invisible to all other classes |
| **package** | ~ | (blank) | Visible to all classes in the package |
| **protected** | # | protected | Like package, but also includes subclasses |
| **ambiguous** | (blank) | N/A | Designer does not know |

## Our Student Record




## Is this Good Design?

- We can create instances and assign values:

```
Student bob = new Student();
bob.name = "Bob";
bob.age = 20;
bob.gpa = 3.1;
```

- But we can also do things that don't make sense:

```
bob.age = -9999;
```

- Maybe we'd like to enforce some constraints (code)

## Information Hiding

- We could replace direct access to age via methods that allow us to run code instead of having direct access to the variable.
- **Mutator** method:

```
public void setAge(int newAge) {
    if(newAge < 0) {
            throw new IllegalArgumentException("Age
                    must be positive");
    }
    age = newAge;
}
```

## Enforcement

- Now we can get a runtime error (Exception) if we do something like:

```
bob.setAge(-9999);
```

- But we haven't eliminated the ability to just set age directly as we did before
- We can change the visibility of fields to allow the compiler to enforce using `setAge()` instead of assignment (=)

## New Student Class

```
class Student {
        public String name;
        private int age;
        public float gpa;

        public void setAge(int newAge) {
                if(newAge < 0) {
                        throw new IllegalArgumentException("Age
                                must be positive");
                }
                age = newAge;
        }

}
```

## Private Visibility

```
class StudentTest {
	public static void main(String[] args)  {
		Student bob = new Student();
		bob.setAge(20);
		System.out.println(bob.age);
	}
}
```

```
$ javac StudentTest.java
StudentTest.java:7: error: age has private access in Student
		System.out.println(bob.age);
```

## Accessor Method

- We will add an additional accessor method to get the value of age for us:

```
public int getAge() {
	return age;
}
```

- It is generally considered poor practice to have `public` modifiable fields in a class
  - Constants might be public, but variables shouldn't be

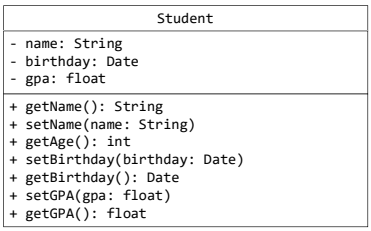
## Further Improvement

- Age is a time-dependent value
  - It would be odd to store it, since it changes
- What doesn't change is birthday
  - But then you have to calculate age rather than store it
- That's okay – we can hide that computation in the implementation of getAge()
  - No caller will know the implementation changed

## Methods in UML

- Methods have up to four components, two of which are required:
  - *Visibility*: optional
  - *Name*: required
  - *Parameters*: optional, though parentheses are required. Each parameter, if present, has a name followed by a colon followed by its type
  - Return type: optional, though if present, preceded by a colon

## Better Student Record




## Object Construction

- All fields in Java are initialized when an instance is created to their type's defaults:

| Type | Default |
|---|---|
| byte/short/int/long | 0 |
| float/double | 0.0 |
| boolean | false |
| char | '\0' |
| Reference | null |

## Constructor

```
public Student(String n, int a, float g) {
      age = a;
      name = n;
      gpa = g;
}
```

- Allows us to build an instance with non-default values without having to call our mutators (which might not even exist for some fields).

## Constructor Types

- **No-arg constructor** – The constructor doesn't take any arguments, so must set the fields to constants or random values, etc.
- **Default constructor** – A no-arg constructor that we didn't have to write. Java creates us one automatically in every class.
- If we write any explicit constructor, we lose the default constructor
  - If we still want a no-arg constructor, we must make it ourselves

## Overloading Constructors

- If we have our explicit constructor and write a no-arg constructor, we'd have two constructors
  - Two or more methods or constructors with the same name are called **overloaded** methods/constructors
- Overloading of methods and constructors is legal if they have different **signatures**:
  - Signature is name plus parameter count and types
  - Return type and modifier list (access specifier/static/etc.) aren't considered part of the signature

## DRY: Don't Repeat Yourself

- We have always tried to avoid repeating code in multiple places
- When we identified code that was in two or more places, we tried to place it in methods
  - If it wasn't identical, we parameterized it to try to make it apply in a more general way
- We always want to avoid **duplication** of code
  - Easier to find, change, and fix if it's all in one place

## DRY: Constructors

- Parameterized Constructor:

```
public Student(String n, int a, float g) {
       age = a;
       name = n;
       gpa = g;
}
```

- No-args Constructor:

```
public Student() {
       age = 17;
       name = "Default";
       gpa = 0.0;
}
```

## `this`

- We can use the keyword `this` to have one constructor call another:

```
public Student(String n, int a, float g) {
       age = a;
       name = n;
       gpa = g;
}

public Student() {
       this("Default", 17, 0.0);
}
```

## this

- In the broader context, `this` represents a reference to the current object (instance)
- With it, we can resolve the idea that OOPLs are not "special" from a computing persepective:
  - `subject.verb(parameter)` => `verb(subject, parameter)`
  - Every instance (non-static) method takes an implicit zero-th parameter whose name is `this`
  - Python makes this explicit, and calls it `self`

## this

- One of the most common uses of `this` outside of calling another constructor on the object is to resolve a scope ambiguity issue:

```
public Student(String n, int a, float g) {
	age = a;
	name = n;
	gpa = g;
}
```

## this

- One of the most common uses of `this` outside of calling another constructor on the object is to resolve a scope ambiguity issue:

```
public Student(String name, int age, float gpa) {
	age = age;   // Always refers to the variable declared
	name = name; // in the nearest enclosing block.
	gpa = gpa;   // So it doesn't work
}
```

## this

- One of the most common uses of `this` outside of calling another constructor on the object is to resolve a scope ambiguity issue:

```
public Student(String name, int age, float gpa) {
	this.age = age;
	this.name = name;
	this.gpa = gpa;
}
```