Need help?

Email: zfk@andrew.cmu.edu (Appointments)
Office: 1004
Phone: 4454-8496
Office Hours: Sunday, 2:00 PM – 3:00 PM
Tuesday, 2:00 PM – 3:00 PM
Open door policy!
Object-Oriented Programming (OOP)
name = "Hammoud";
gpa = 3.6;
year = 3;
courses = ["15440", "15230"];

function canGraduate(gpa, year, courses) {
    return if (year > 2 && gpa > 2);
}

Student = {
    name = "Hammoud";
    gpa = 3.6;
    year = 3;
    courses = ["15440", "15230"];

    canGraduate: function() {
        return if (this.year > 2 &&
            this.gpa > 2);
    }
}

Student.canGraduate();
OOP

PROCEDURAL PROGRAMMING

- You have methods that operate on data
- Methods and attributes/data are decoupled

OBJECT-ORIENTED PROGRAMMING

- Combine group of related functions and variables into a unit (object)
- Function does not have any inputs (because these inputs are encapsulated in the students object)
- Methods and attributes/data are coupled
A programming paradigm based on objects
A programming paradigm based on objects

Encapsulation  Inheritance  Polymorphism  Abstraction
Objects
A class groups attributes/data and methods

```java
public class Student {
    String name;
    Double gpa;
    Integer year;

    public String canGraduate() {
        return if (year > 2 && gpa > 2);
    }
}
```

**OOP (objects)**

A class is the blueprint from which individual objects are created

Access modifiers describe the accessibility (scope) of data
OOP (objects)

A class groups attributes/data and methods

```java
public class Student {
    String name;
    Double gpa;
    Integer year;

    public Boolean canGraduate(){
        return if (year > 2 && gpa > 2);
    }
}
```

Instantiate objects

```java
Student s1 = new Student();
Student s2 = new Student();
Student s3 = new Student();
```
Constructors
A class groups attributes/data and methods

```java
public class Student {
    String name;
    Double gpa;
    Integer year;

    public Boolean canGraduate()
    {
        return if (year > 2 &&
                gpa > 2);
    }
}
```

We need to set attributes when we instantiate objects!
A class groups attributes/data and methods

```java
public class Student {
    String name;
    Double gpa;
    Integer year;
    public Student() {
    }
    public Boolean canGraduate(){
        return if (year > 2 &&
            gpa > 2);
    }
}
```

Student s1 = new Student();

Student s2 = new Student();

Student s3 = new Student();
A class groups attributes/data and methods

```java
public class Student {
    String name;
    Double gpa;
    Integer year;

    public Student(String name) {
    }

    public Boolean canGraduate() {
        return year > 2 && gpa > 2;
    }
}
```

```java
Student s1 = new Student();
Student s2 = new Student();
Student s3 = new Student();
```
A class groups attributes/data and methods

```java
public class Student {
    String name;
    Double gpa;
    Integer year;

    public Student(String name) {
        name = name;
    }

    public Boolean canGraduate() {
        return year > 2 && gpa > 2;
    }
}
```

```
Student s1 = new Student();
Student s2 = new Student();
Student s3 = new Student();
```
A class groups attributes/data and methods

```java
public class Student {
    String name;
    Double gpa;
    Integer year;

    public Student(String name) {
        this.name = name;
    }

    public Boolean canGraduate() {
        return if (year > 2 && gpa > 2);
    }
}
```

Student s1 = new Student();
Student s2 = new Student();
Student s3 = new Student();
A class groups attributes/data and methods.

```java
public class Student {
    String name;
    Double gpa;
    Integer year;
    
    public Student(String name, Double gpa) {
        this.name = name;
    }
    
    public Boolean canGraduate(){
        return if (year > 2 && gpa > 2);
    }
}
```

OOP (constructors)

```java
Student s1 = new Student();
Student s2 = new Student();
Student s3 = new Student();
```
A class groups attributes/data and methods.

```java
public class Student {
    String name;
    Double gpa;
    Integer year;

    public Student(String name, Double gpa) {
        this.name = name;
        gpa = gpa;
    }

    public boolean canGraduate() {
        return if (year > 2 && gpa > 2);
    }
}
```

Example:

```java
Student s1 = new Student();
Student s2 = new Student();
Student s3 = new Student();
```
A class groups attributes/data and methods

```java
public class Student {
    String name;
    Double gpa;
    Integer year;

    public Student(String name, Double sGpa) {
        this.name = name;
        gpa = sGpa;
    }

    public Boolean canGraduate(){
        return if (year > 2 && gpa > 2);
    }
}
```

```java
Student s1 = new Student("snoopy", "2.1");
Student s2 = new Student("popeye", "4.0");
Student s3 = new Student("Goofy", "3.6");
```
A class groups attributes/data and methods

```java
public class Student {
    String name;
    Double gpa;
    Integer year;

    public Student(String name, Double gpa) {
        this.name = name;
        gpa = gpa;
    }

    public Boolean canGraduate() {
        return if (year > 2 &&
                   gpa > 2);
    }
}
```

---

Student s1 = new Student("snoopy", "2.1");

Student s2 = new Student("popeye", "4.0");

Student s3 = new Student("Goofy", "3.6");

---

What happens if we don’t use “this”?
A class groups attributes/data and methods

```java
public class Student {
    String name;
    Double gpa;
    Integer year;

    public Student(String name, Double sGpa) {
        name = name;
        gpa = sGpa;
    }

    public Boolean canGraduate(){
        return if (year > 2 && gpa > 2);
    }
}
```

What happens if we don’t use “this”?

Student s1 = new Student(“snoopy”, ”2.1”);
Student s2 = new Student(“popeye”, ”4.0”);
Student s3 = new Student(“Goofy”, ”3.6”);
OOP (constructors)

```java
Student s1 = new Student("snoopy", "2.1");
Student s2 = new Student("popeye", "4.0");
Student s3 = new Student("Goofy", "3.6");
```

s1.canGraduate();
s2.canGraduate();
s3.canGraduate();
Encapsulation
**OOP** (encapsulation)

### PROCEDURAL PROGRAMMING

```javascript
name = "Hammoud";
gpa = 3.6;
year = 3;
courses = ["15440", "15230"];

function canGraduate(gpa, year, courses) {
    return if (year > 2 && gpa > 2);
}
```

### OBJECT-ORIENTED PROGRAMMING

```javascript
Student = {
    name = "Hammoud";
gpa = 3.6;
year = 3;
courses = ["15440", "15230"];

canGraduate: function() {
    return if (this.year > 2 &&
              this.gpa > 2);
}
}

Student.canGraduate();
```
OOP (encapsulation)

Student = {
    name = "Hammoud";
    gpa = 3.6;
    year = 3;
    courses = ["15440", "15230"];
    canGraduate: function() {
        return if (this.year > 2 && this.gpa > 2);
    }
}
Student.canGraduate();
Restricting access to an object’s components

So how can we change an attribute as we encapsulate?
public class Student {

    String name;
    Double gpa;
    Integer year;

    public Student(name, sGpa) {...}

    public String canGraduate(){...}

    public void setName(String newName) {
        this.name = newName;
    }

    public String getName() { return name }
}

Student s1 = new Student(“snoopy”, ”2.1”);
s1.setName(“Goofy”);
Inheritance
public class Student {
    String name; Double gpa; Integer year;
    public Student() {
    }
    public String canGraduate() {
    }
}

public class CrossStudent {
    String name; Double gpa; Integer year; String university;
    public Student() {
    }
    public String canGraduate() {
    }
}

public class IntStudent {
    String name; Double gpa; Integer year; String origin;
    public Student() {
    }
    public String canGraduate() {
    }
}
OOP (inheritance)

public class Student {
    String name; Double gpa;
    Integer year;
    public Student() {
    }
    public String canGraduate(){
    }
}

Redundant code!

public class CrossStudent {
    String name; Double gpa;
    Integer year; String university;
    public Student() {
    }
    public String canGraduate(){
    }
}

public class IntStudent {
    String name; Double gpa;
    Integer year;
    String origin;
    public Student() {
    }
    public String canGraduate(){
    }
}
public class Student {
    String name; Double gpa;
    Integer year;
    public Student() {
    }
    public String canGraduate() {
    }
}

public class CrossStudent extends Student {
    String university;
}

public class IntStudent extends Student {
    String origin;
}
OOP (inheritance)

public class Student {
    String name; Double gpa;
    Integer year;
    public Student() {
    }
    public String canGraduate(){
    }
}

public class CrossStudent extends Student {
    String university;
}

public class IntStudent extends Student {
    String origin;
}

This introduces polymorphism (discussed in the next section)
OOP (inheritance)

```
Student
  Name
  GPA
  Year
  canGraduate()

Local
CrossRegistered
International
```

Superclass

Subclass
OOP (inheritance)

- Organizes related classes in a hierarchy, which allows for reusability and extensibility of common code
- Subclasses extend the functionality of a superclass
- Subclasses inherit all the methods of the superclass (excluding constructors and privates)
- Subclasses can override methods from the superclass (more on this later)
Polymorphism
Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.
Polymorphism (example)

```java
public class Student {
    String name; Double gpa;
    Integer year;
    public Student() {
    }
    public Boolean canGraduate(){
    }
}

public class CrossStudent extends Student {
    String university;
    Integer grade_level;
    public Boolean canGraduate() {
        return (grade_level > 3)
    }
}

public class IntStudent extends Student {
    String origin;
    Boolean pass;
    public Boolean canGraduate() {
        return pass;
    }
}
```
Access Control
# OOP (Access control)

Access modifiers describe the accessibility scope of data, methods and constructors.

<table>
<thead>
<tr>
<th>Public</th>
<th>Protected</th>
<th>Private</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allows the access of the object/attributes/methods from any other program that is using this object.</td>
<td>You can use objects, attributes or methods with protected access modifier if within the same class, within the same package, or referenced from a subclass.</td>
<td>Restricted than a protected variable: you can use attributes, variables and methods only in the same class.</td>
</tr>
</tbody>
</table>

```java
public String name;
public String getName() { … }
private Student(String name, int sAge) { … }
```
OOP (Access control)

Public

Allows the access of the object/attributes/methods from any other program that is using this object

```java
public class Student {
    public void setName(String newName) {
        this.name = newName;
    }
}

class Test {
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.setName("Snoopy");
    }
}
```
OOP (Access control)

public class Student {
    private void setName(String newName) {
        this.name = newName;
    }
}

public class Test {
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.setName("Snoopy");
    }
}

Private

Restricted than a protected variable: you can use attributes, variables and methods only in the same class.
OOP (Access control)

Protected

You can use objects, attributes or methods with protected access modifier if within the same class, within the same **package**, or referenced from a subclass.

```
package BMW;

public class BMW_i8 {
  ...
}
```
Object & Class Variables
Object Variables
When each object has its own variable/attribute. Such as student name, gpa, and year.

Class/static Variables
When an attribute should describe an entire class of objects instead of a specific object. For example, all students live on Earth. There's only one copy of this variable for the entire class.
public class Student {
    public static final String currentPlanet = "EARTH";
}

public class Test() {
    public static void main(String[] args) {
        Student Goofy = new Student();
        String planet = Goofy.currentPlanet;
    }
}

Class/static Variables
When an attribute should describe an **entire class** of objects instead of a specific object. For example, all students live on Earth. There’s only one copy of this variable for the entire class
Overloading
Overloading Methods

• Methods overload one another when they have the same method name but:
  • The **number of parameters** is different for the methods
  • The parameter **types** are different

```java
public void changeDate(int year) {
    // do cool stuff here
}

public void changeDate(int year, int month) {
    // do cool stuff here
}

public void addSemesterGPA(float newGPA) {
    // process newGPA
}

public void addSemesterGPA(double newGPA) {
    // process newGPA
}
```
Overloading Methods

```java
public void changeDate(int year) {
    // do cool stuff here
}

public void changeDate(int month) {
    // do cool stuff here
}
```

How about this?

We can't overload methods by just changing the parameter name!
Overriding
**Overriding** (methods)

```java
public class ClassA {
    public Integer someMethod() {
        return 3;
    }
}

public class ClassB extends ClassA {
    // this is method **overriding**:
    public Integer someMethod() {
        return 4;
    }
}
```

The same method specifications (name and type) but different implementation
Overriding (Object class)

- Any class extends the **Java** superclass “**Object**”.
- The Java “**Object**” class has 3 important methods:
  - public boolean equals(Object obj);
  - public int hashCode();
  - public String toString();
- The hashCode is just a number that is generated by any object:
  - It **shouldn’t** be used to compare two objects!
  - Instead, **override** the equals, hashCode, and toString methods.
Overriding  

Example: **Overriding** the `toString` and `equals` methods in our `Student` class:

```java
public class Student {
    ...
    public String toString() {
        return this.name;
    }
}
```
Overriding (Object class)

Overriding the `toString` and `equals` methods in our `Student` class:

```java
public class Student {
    ... 
    public boolean equals(Object obj) {
        if (obj.getClass() != this.getClass())
            return false;
        else {
            Student s = (Student) obj;
            return (s.getName().equals(this.name));
        }
    }
}
```
Abstract Classes
Abstract Classes

• A class that is **not completely implemented**.
• Contains one or more *abstract* methods (methods with no bodies; *only signatures*) that subclasses must implement
• Cannot be used to instantiate objects
• Abstract class header:
  
  ```java
  accessModifier abstract class className
  public abstract class Car
  ```

• Abstract method signature:

  ```java
  accessModifier abstract returnType methodName (args);
  public abstract int max_speed ();
  ```

• Subclass signature:

  ```java
  accessModifier class subclassName extends className
  public class Mercedes extends Car
  ```
Interfaces
Interfaces

• A special abstract class in which all the methods are abstract
• Contains only abstract methods that subclasses must implement
• Interface header:

  accessModifier interface interfaceName
  public interface Car

• Abstract method signature:

  accessModifier abstract returnType methodName ( args );
  public abstract String CarType ( args );

• Subclass signature:

  accessModifier class subclassName implements someInterface
  public class BMW implements Car
Generic methods
Generic Methods

Generic or parameterized methods receive the data-type of elements as a parameter

E.g.: a generic method for sorting elements in an array (be it Integers, Doubles, Objects etc.)
A Simple Box Class

A *generic class* is defined with the following format:

```java
class my_generic_class <T1, T2, ..., Tn>
{
    /* ... */
}
```

*Type parameters*
A Simple Box Class

Now to make our Box class generic:

```java
public class Box<T> {
    // T stands for "Type"
    private T t;
    public void set(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
}
```

• To create, for example, an Integer “Box”:

```java
Box<Integer> integerBox;
```
Java Generic Collections

• Classes that represent data-structures
  - *Generic or parameterized* since the elements’ data-type is given as a parameter
  - E.g.: LinkedList, Queue, ArrayList, HashMap, Tree

• They provide methods for:
  - Iteration
  - Bulk operations
  - Conversion to/from arrays
Bank Example
(Refer to the attached code)