# Events this week

<table>
<thead>
<tr>
<th>Event</th>
<th>Date</th>
<th>Time</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resume Editing Drop-In Session</td>
<td>Mon., Feb 1</td>
<td>11 am – 2 pm</td>
<td>Rm 255, ICICS/CS</td>
</tr>
<tr>
<td>EADS Info Session</td>
<td>Mon., Feb 1</td>
<td>3:30 – 5:30 pm</td>
<td>CEME 1202</td>
</tr>
<tr>
<td>Job Interview Practice Session (for non-coop students)</td>
<td>Tues., Feb 2</td>
<td>11 am – 1 pm</td>
<td>Rm 206, ICICS/CS</td>
</tr>
</tbody>
</table>

# Events next week

<table>
<thead>
<tr>
<th>Event</th>
<th>Date</th>
<th>Time</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>RIM Info Session</td>
<td>Thurs., Feb 4</td>
<td>5:30 – 7 pm</td>
<td></td>
</tr>
<tr>
<td>Finding a Summer Job or Internship Info Session</td>
<td>Wed., Feb 10</td>
<td>12 pm</td>
<td></td>
</tr>
<tr>
<td>Masters of Digital Media Program Info Session</td>
<td>Thurs., Feb 11</td>
<td>12:30 – 1:30 pm</td>
<td></td>
</tr>
</tbody>
</table>

## Administrivia

- Lecture slides (day by day) are on the web:
- Assignment #2 is out
  - Due Thursday February 11, 10:00pm
Where are we?

- We are discussing class design – particularly with reference to inheritance.
  - Use overloading with care
  - Open-closed principle
    - Open for extension but closed for modification
  - is-a Inheritance
  - Preconditions and postconditions
  - Worry about mutable superclasses (Rectangle-Square)
  - Delegation

Multiple Inheritance

- Multiple inheritance occurs when a class has more than one super-class.
- Multiple inheritance is supported by some programming languages (e.g., C++) but not others (e.g., Java).
- Multiple inheritance can lead to problems, for example, the classic *diamond* problem:

```
Suppose Person has a method myMethod() that's overridden in a different way in Student and Employee and that's not overridden in TeachingAssistant. Which version of the method should the following code call:

TeachingAssistant ta = new TeachingAssistant();
ta.myMethod();
```
Handling Multiple Inheritance in Java

- We can use interfaces and delegation to implement multiple class inheritance if necessary
- For instance:
  instead of this: you can do this:

```
Student           Employee
              ^        ^
  StudentInterface EmployeeInterface

Multiple Inheritance Example

```interface StudentInterface {
   public float getGPA();
}
```interface EmployeeInterface {
   public float getSalary();
}
```public class Student implements StudentInterface {
   protected float GPA;
   public float getGPA() {
      // code for getGPA
   }
}
public class Employee implements EmployeeInterface {
    protected float salary;
    public float getSalary() {
        // code for getSalary
    }
}

public class TeachingAssistant implements StudentInterface, EmployeeInterface {
    private Student student;
    private Employee employee;

    public TeachingAssistant() {
        student = new Student();
        employee = new Employee();
    }

    public float getGPA() {
        return student.getGPA();
    }

    public float getSalary() {
        return employee.getSalary();
    }
}
Multiple Inheritance Example (continued)

- Does this resolve the “diamond” problem?
- What if both Student and Employee:
  - Implement `myMethod()`? ✓
  - Have an attribute named `id`? ✓

Name Collisions Among Interfaces

- A Java class may extend another class and implement one or more interfaces
- Inherited method from one interface may have same name as a method in another class or interface
- Name Collision procedure:
  - if methods have different signatures, they are considered overloaded
  - if they have same signature and return type, they are one method
  - if they have same signature but different return types, produce compilation error
  - if they have same signature and return type, but throw different exceptions, they are one method that throws the intersection of the exceptions thrown by each of them
General Design Guidelines for Inheritance

• Place common attributes and methods in the superclasses
• Use inheritance to model only *is-a* type relationships
• Use abstract classes and interfaces to design extensible families of objects with common properties
  • e.g., employees of different types
  • e.g., different types of objects to be drawn in a CAD application

Exercise

Which is the right way to define ellipse and circle?
Key Concepts In This Lecture

• There are a lot of related concepts we covered in the last week
  • When you design a superclass, think about whether it might be extended in the future (i.e., which methods should be protected instead of private, etc.). This is the open-closed principle in action.
  • In Java, a subclass is considered a subtype as is an implementation of an interface. To ensure an instance of a subclass (or a class that implements an interface) is substitutable for its superclass (or its interface) we need to follow the Liskov Substitutability Principle (LSP). i.e., watch out that pre-conditions and post-conditions of overridden methods do the right thing.
  • If we want to reuse code but can’t do it via a subclass because we’d violate the LSP, we can use delegation where we keep an object of the type from which we want the code and we call the object’s methods to do the work we want done.
  • If we want one class to include behavior from different types, use interfaces (and sometimes delegation too!)

Class Design Review
TicketWizard Example

Learning Objectives

- Given a problem statement and a design, identify errors and violations of design principles in the design
- Given a problem statement and a wrong design correct the design and create an appropriate UML class diagram for the problem
A TicketWizard Office needs a software system to track various events, their venues, and ticket orders for the events.

- Each event has a name, description, date, time, a base ticket price and occurs at a single venue.
- Each venue has a name, address, phone number.
- Different events can have different seating plans. The seating plan consists of a number of sections and each section contains a number of seats. The price of a seat is determined by the base ticket price of the event and the section’s price factor. A venue may host many different events, one event at a time, of course.

Customers can place orders, which are made up of one or more seats for one or more events. Ticket office employees can also place orders; they enjoy a 10% discount on any regular ticket price.

Customers can pay for their orders by cash or charge them to a credit card. For each order, the system must track the type of payment.

Finally, the system must track customer information so that customers can be notified if the event is changed or cancelled.
Some Issues to consider

- Does a venue need to know about events? If so, how?
- Does an event need to know about venue? If so, how?
- Do we need Seat objects?
- Do we need Ticket objects?
- Do we need Customer objects?
- Do we need Employee objects?
- What other objects do we need?

How many errors can you find in this design? 

Note: The default multiplicities are 1