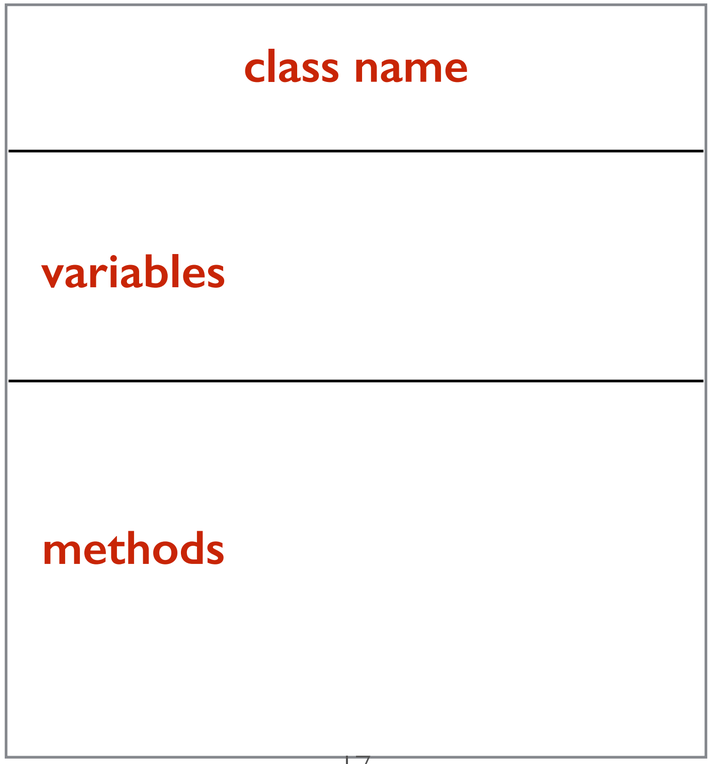
UML

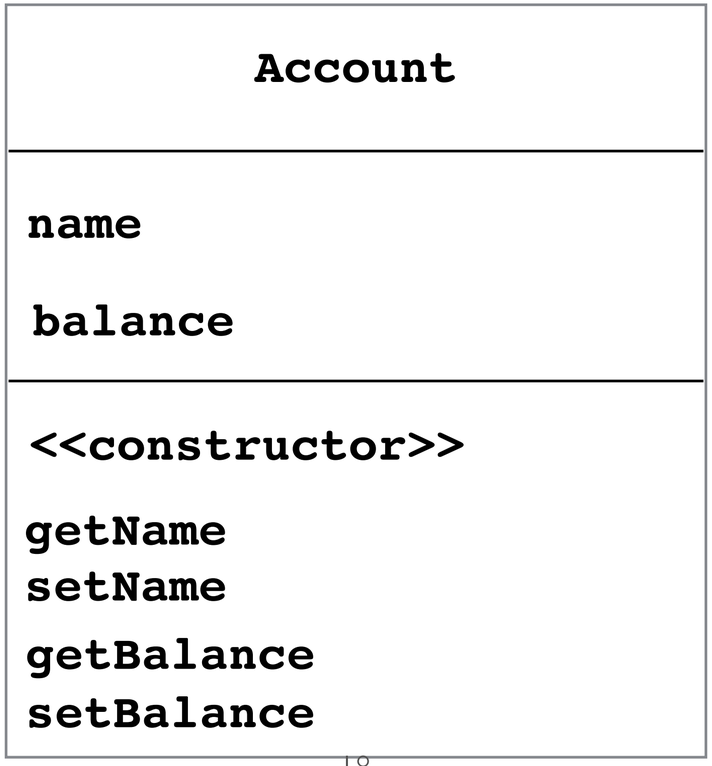
UML

- UML = United Modeling Language
  - Graphical models of object-oriented software.



UML

- Let's look at the Account.java example from the previous class.
- This is a good start, but it needs more information to be useful..



UML

- The - symbol indicates private, and + indicates public.

```
Account
---------------------
-name
-balance
---------------------
<<constructor>>
+getName
+setName
+getBalance
+setBalance
```

UML

- Since it is also important to know the **types** of variables and **parameters** of methods, we add more information..

```
Account
-----------------------------------------------
-name: String
-balance: double
-----------------------------------------------
<<constructor>> Account(name: String, balance: double)
+getName(): String
+setName(name: String)
+getBalance(): double
+setBalance(amount: double)
```

UML

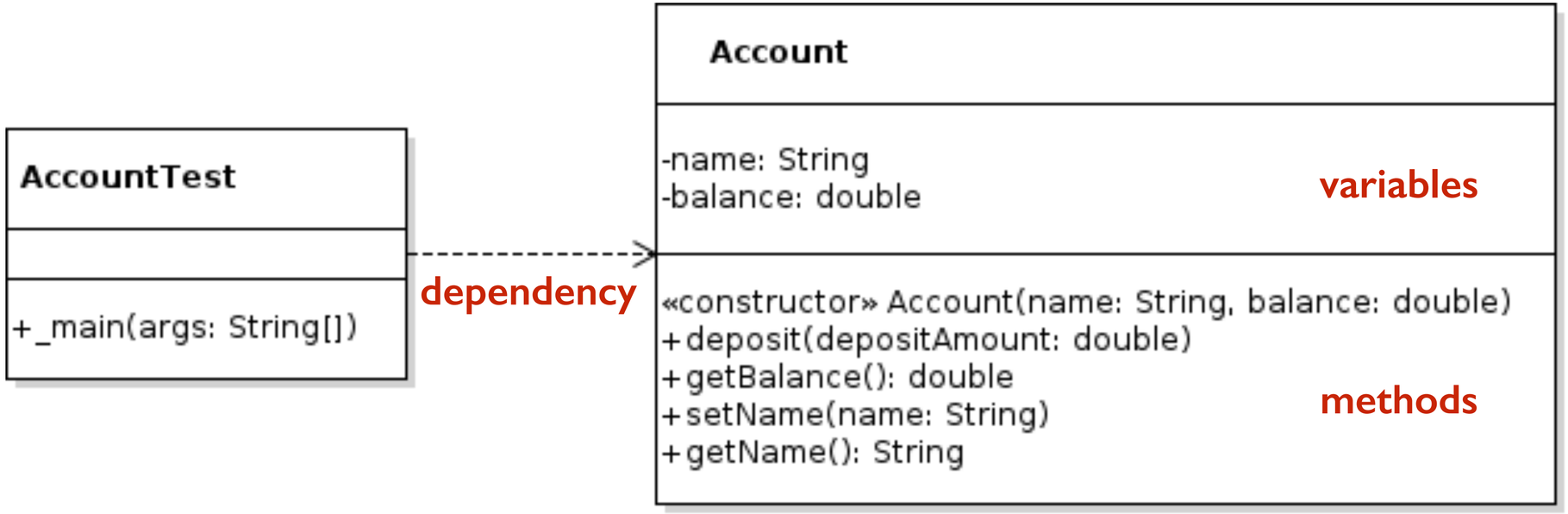
- In this course, we'll use **Violet UML Editor**.
  - Download & directions for installation:
    - *On Blackboard > Content > this week!*

- It's free, cross-platform and easy to use!**

- Let's do a demo!

UML

- **Dependencies** between classes visually show the interaction between these classes.



DEPENDENCY RELATIONSHIP

`ClassA` ⇢ `ClassB`

- The dependency relationship is a generalized connection between two classes.
  - *Do not overuse! Check to see if other relationships are more meaningful.*

- ClassA **depends on** ClassB.
  - *This implies one or more of the following:*
    - At least one ClassB object is referenced in ClassA.
    - There is an import statement in ClassA for ClassB.
    - At least one class method in ClassB is called by ClassA.

UNIDIRECTIONAL ASSOCIATION

`ClassA` ⟶ `ClassB`

- Indicates dependency of only one class on another.

- ClassA **<u>uses and depends on</u>** ClassB, *but* ClassB does not reference ClassA.
  - This implies that:
    - At least one ClassB object is referenced in ClassA.
    - There is an `import` statement in ClassA for ClassB.
    - There is *not* an `import ClassA` statement in ClassB.

UNIDIRECTIONAL ASSOCIATION

`ClassA` ⟶ `ClassB`

example - ***not in our UML format!***

BIDIRECTIONAL ASSOCIATION

```
ClassA ——— ClassB
```

- Indicates codependency of two classes.

- ClassA and ClassB **both depend upon each other**.
  - This implies that:
    - At least one ClassB object is referenced in ClassA.
    - At least one ClassA object is referenced in ClassB.
    - There is an import statement in ClassA for ClassB.
    - There is an import statement in ClassB for ClassA.

BIDIRECTIONAL ASSOCIATION

```
ClassA ——— ClassB
```

example - ***not in our UML format!***

AGGREGATION RELATIONSHIP

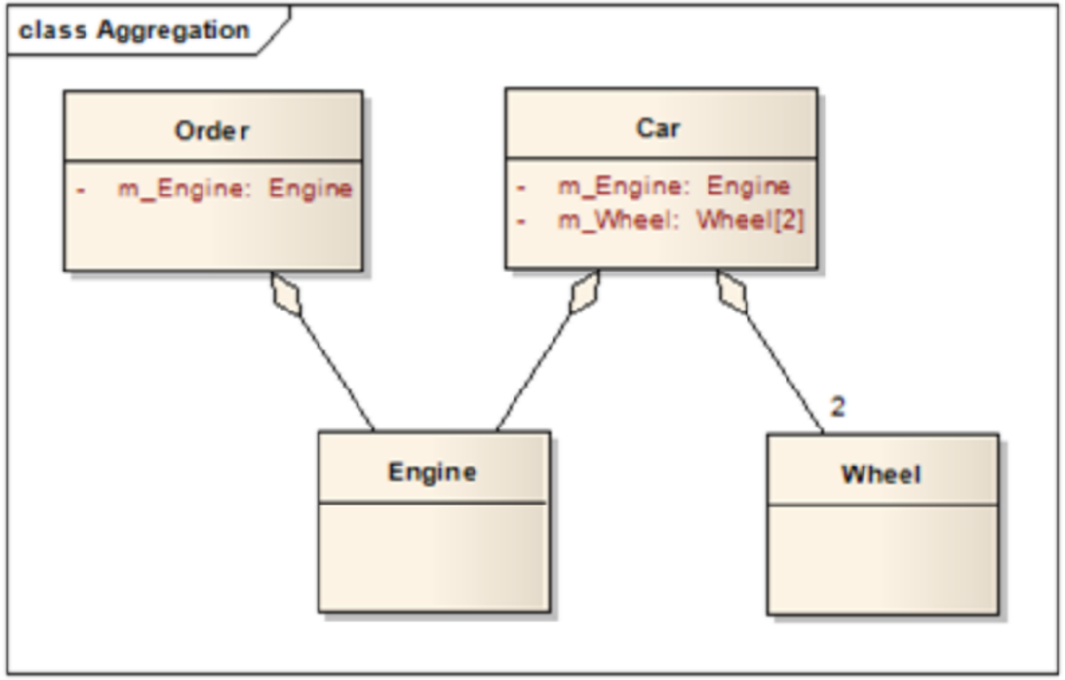
```
ClassA ◇——— ClassB
```

- Indicates shared association.

- ClassA <u>references</u> ClassB, but is not the only reference.
  - ClassA does <u>*not*</u> own ClassB.
  - There is <u>*not*</u> a parent-child relationship between ClassA & ClassB.

AGGREGATION RELATIONSHIP

```
ClassA ◇———— ClassB
```



example

COMPOSITION RELATIONSHIP

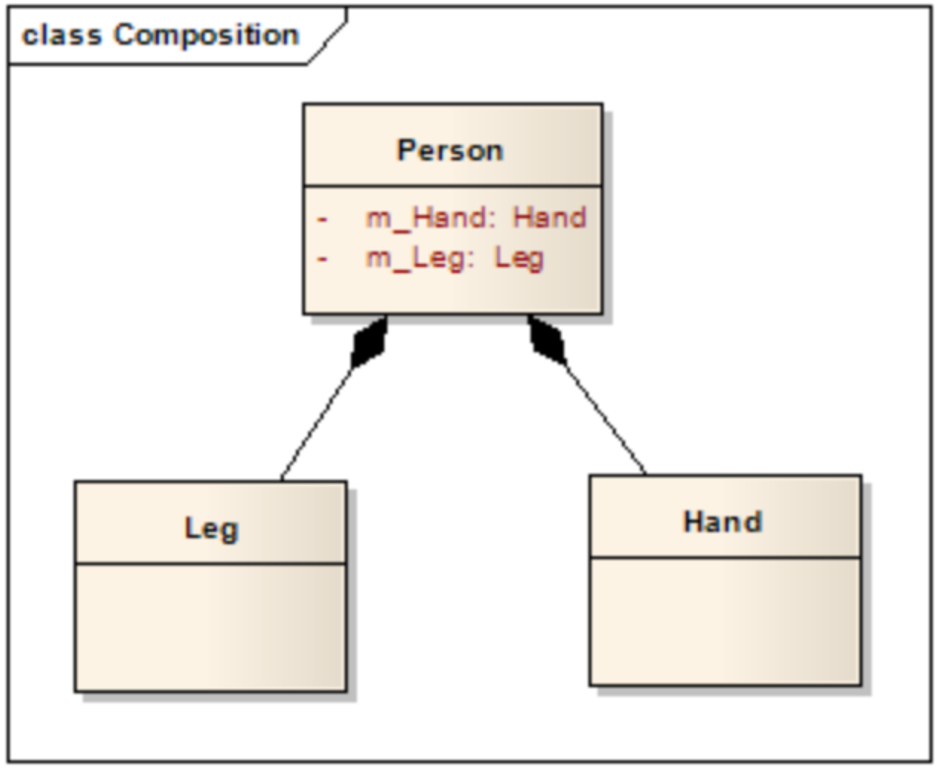
```
ClassA ◆——— ClassB
```

- Indicates not-shared association. Strong dependency.

- ClassA is the container for ClassB.
  - If ClassA is deleted, ClassB will be removed as well.

COMPOSITION RELATIONSHIP

```
ClassA ◆——— ClassB
```



example

REALIZATION RELATIONSHIP

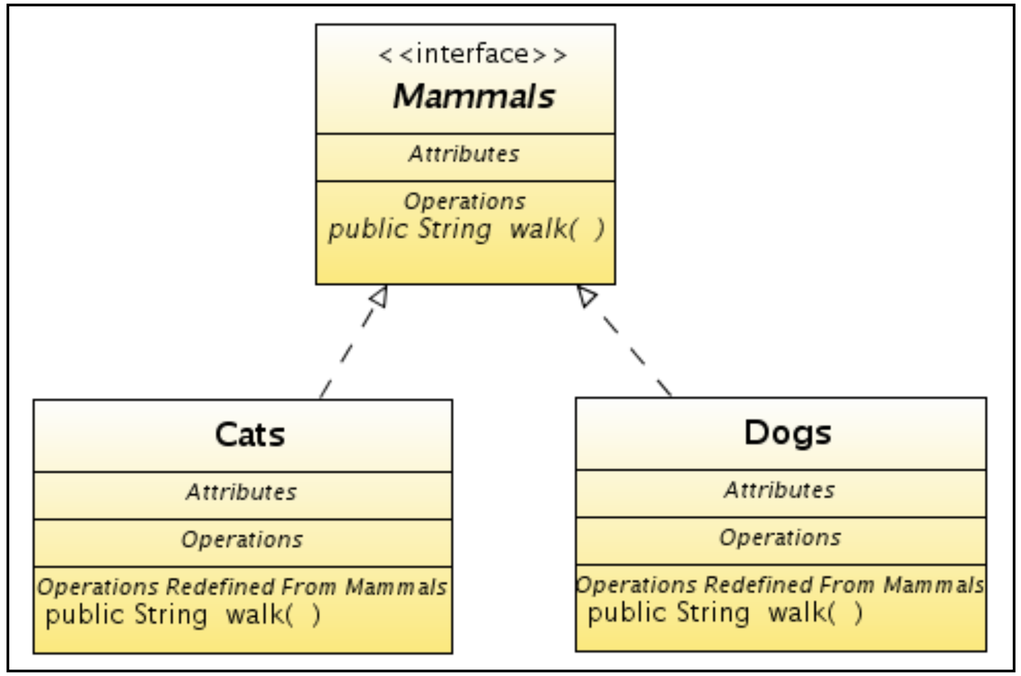
`ClassA` ⇢▷ `ClassB`

- Indicates inheritance, where ClassB is an *interface*.

- ClassA **<u>inherits from</u>** ClassB.
  - Declared in code as ClassA *implements* ClassB.
  - ClassA is the subclass, and ClassB is the interface.

REALIZATION RELATIONSHIP

```
ClassA ·······▷ClassB
```



example - ***not in our UML format!***

GENERALIZATION RELATIONSHIP

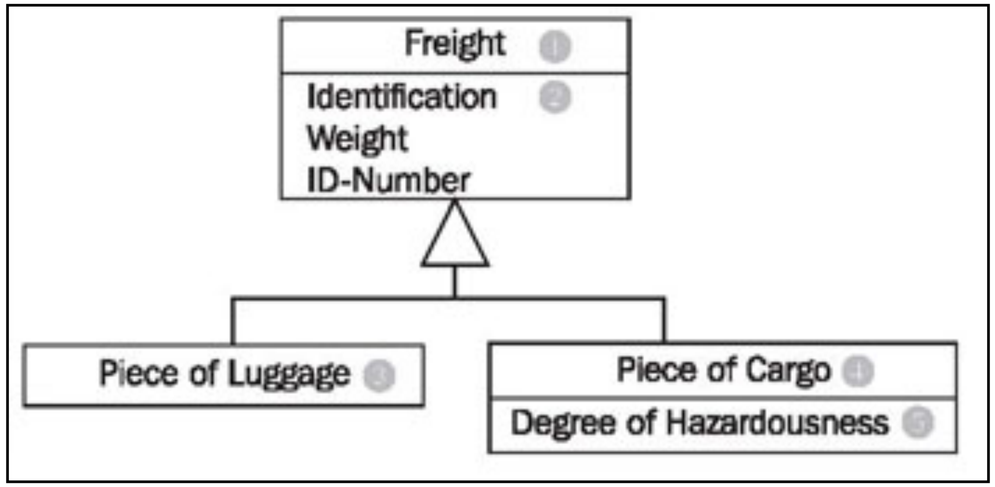
`ClassA` ⟶ `ClassB`

- Indicates inheritance between classes.

- ClassA **<u>inherits from</u>** ClassB.
  - Declared in code as ClassA *extends* ClassB.
  - ClassA is the subclass, and ClassB is the superclass.

GENERALIZATION RELATIONSHIP

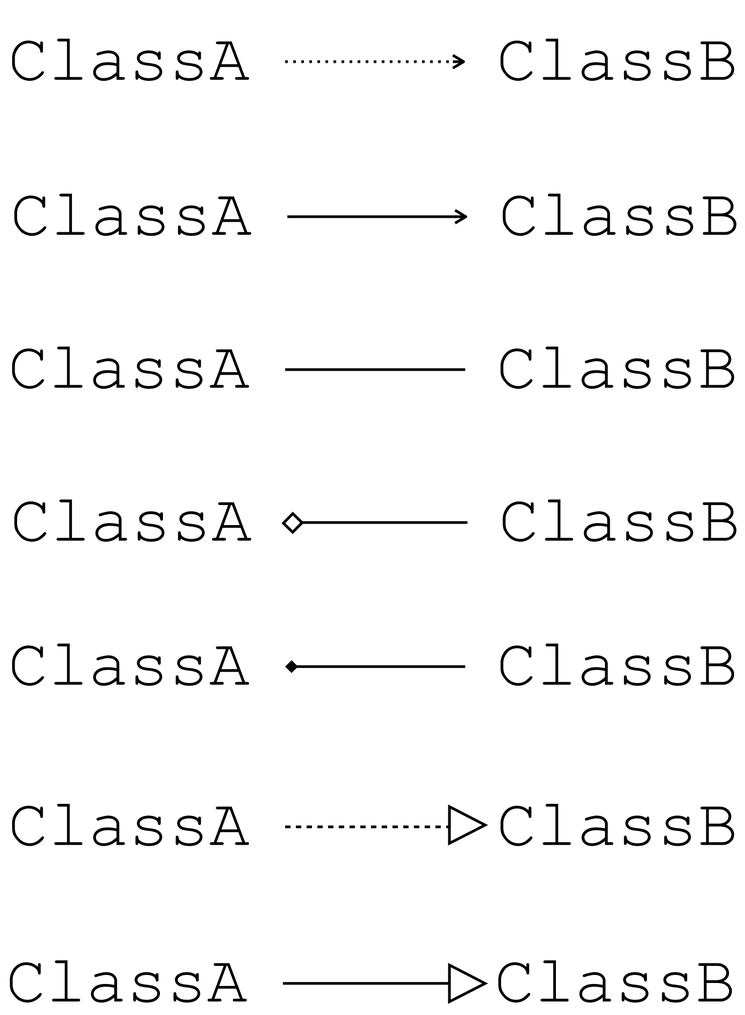
```
ClassA ———▷ClassB
```



*example - **not in our UML format!***

UML

- Dependency Relationship
- Unidirectional Association
- Bidirectional Association
- Aggregation Relationship
- Composition Relationship
- Realization Relationship
- Generalization Relationship



UML RESOURCES

- **`Employee`** example code: Chapter 8

- Want even *more* information on UML?
  - Advanced UML software & concepts: http://www.uml.org
  - In-depth overview of syntax:
    - http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/

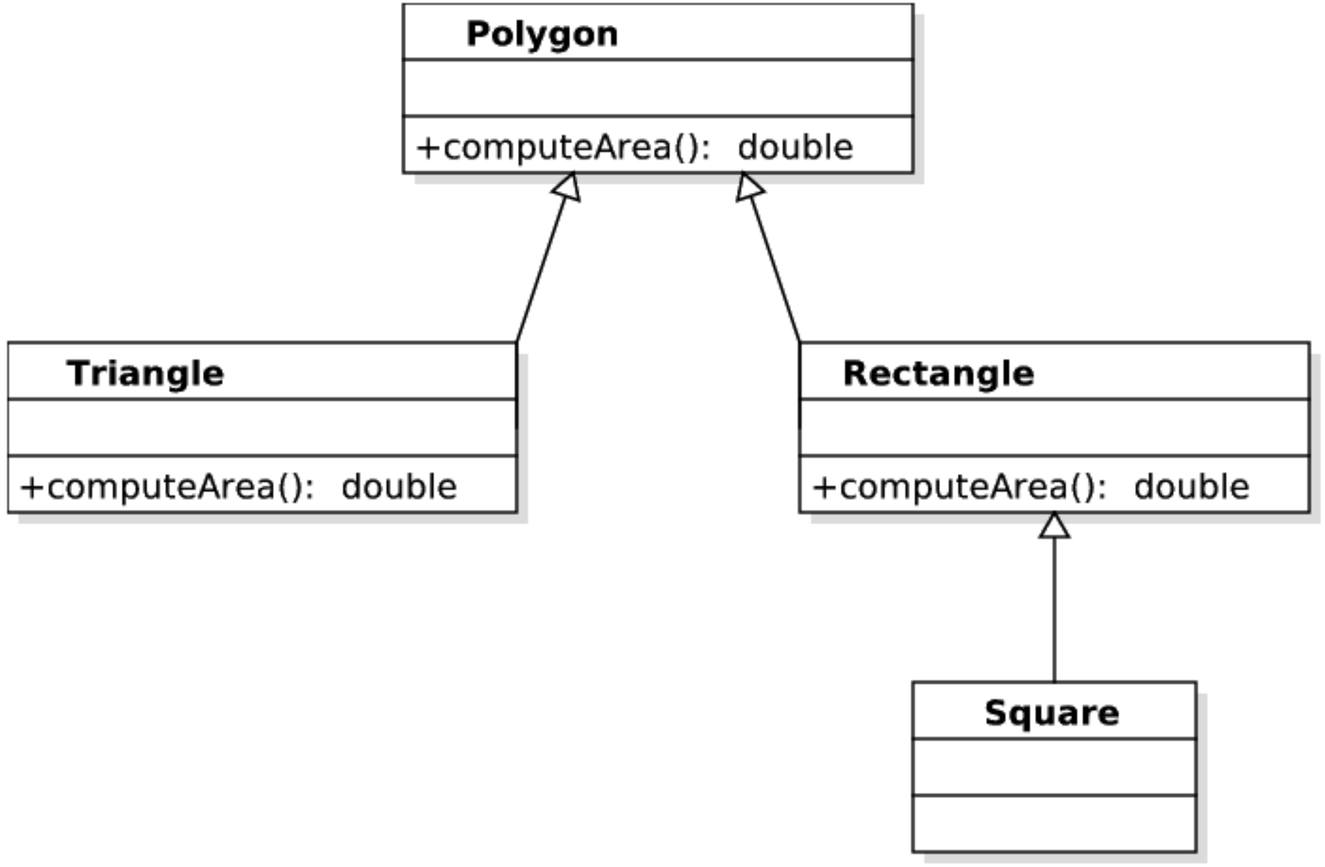
POLYGON EXAMPLE

- A rectangle is a polygon, and a triangle is also a polygon.
- A square is a type of rectangle.
- The area of all of these shapes can be computed.

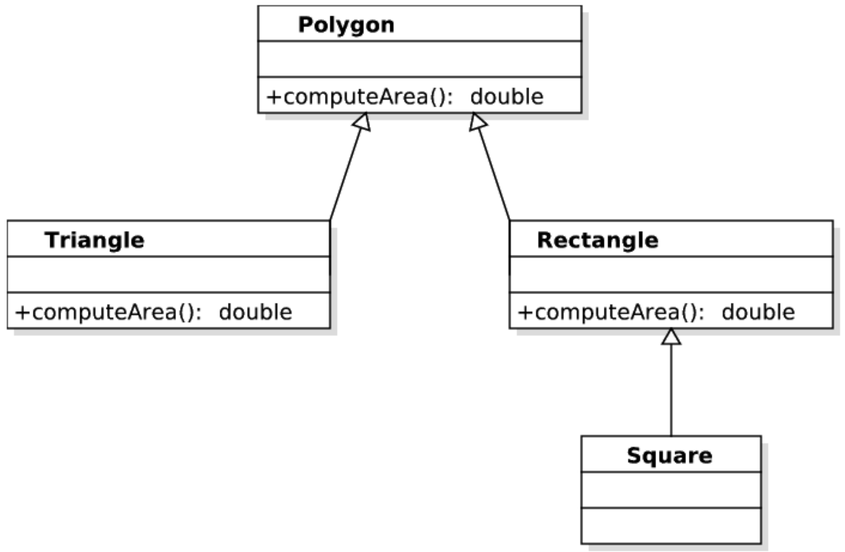
POLYGON EXAMPLE

- A **rectangle** *is a* **polygon**, and a **triangle** *is* also *a* polygon.
- A **square** *is a* type of rectangle.
- The area of all of these shapes can be computed.

POLYGON EXAMPLE

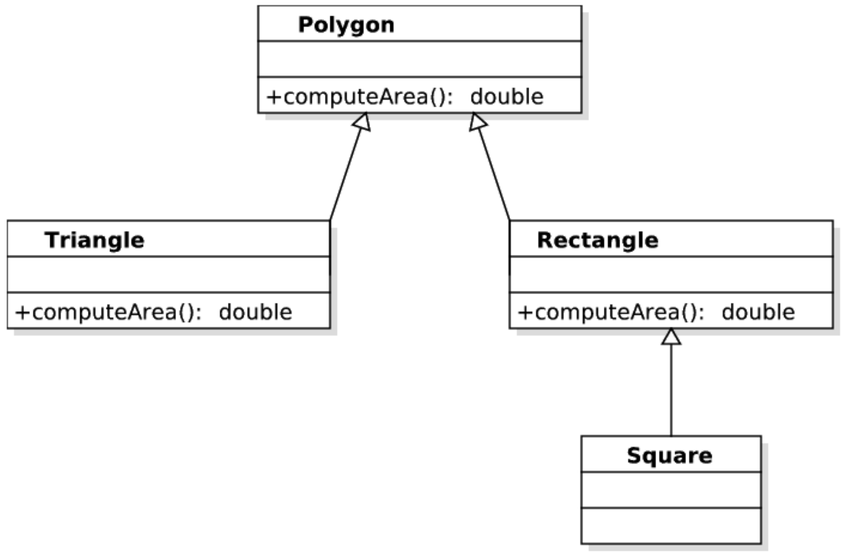


POLYGON EXAMPLE



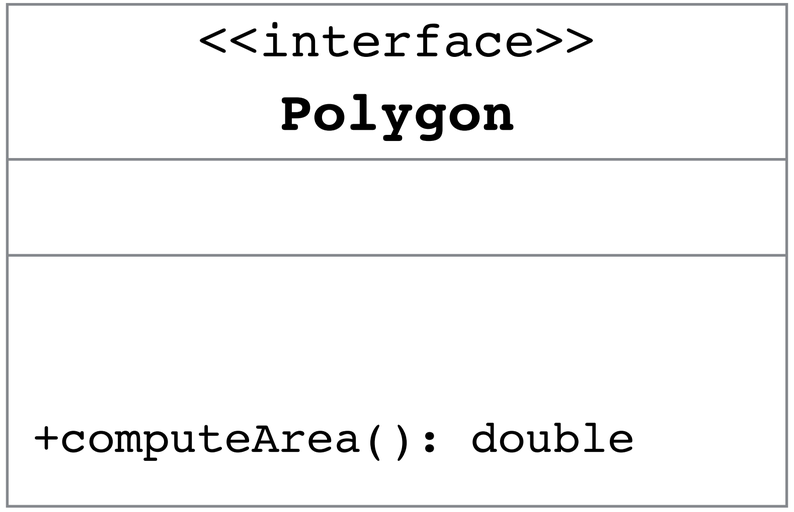
- There are 4 Java classes, and therefore 4 files:
  - Polygon.java
  - Triangle.java
  - Rectangle.java
  - Square.java

POLYGON EXAMPLE



- A Square *is a* Rectangle.
  - **`Square`** *extends* **`Rectangle`**.
- A Triangle *is a* Polygon.
  - **`Triangle`** *extends* **`Polygon`**.
- *Should `Polygon.java` be an abstract class or an interface?*

POLYGON AS AN INTERFACE

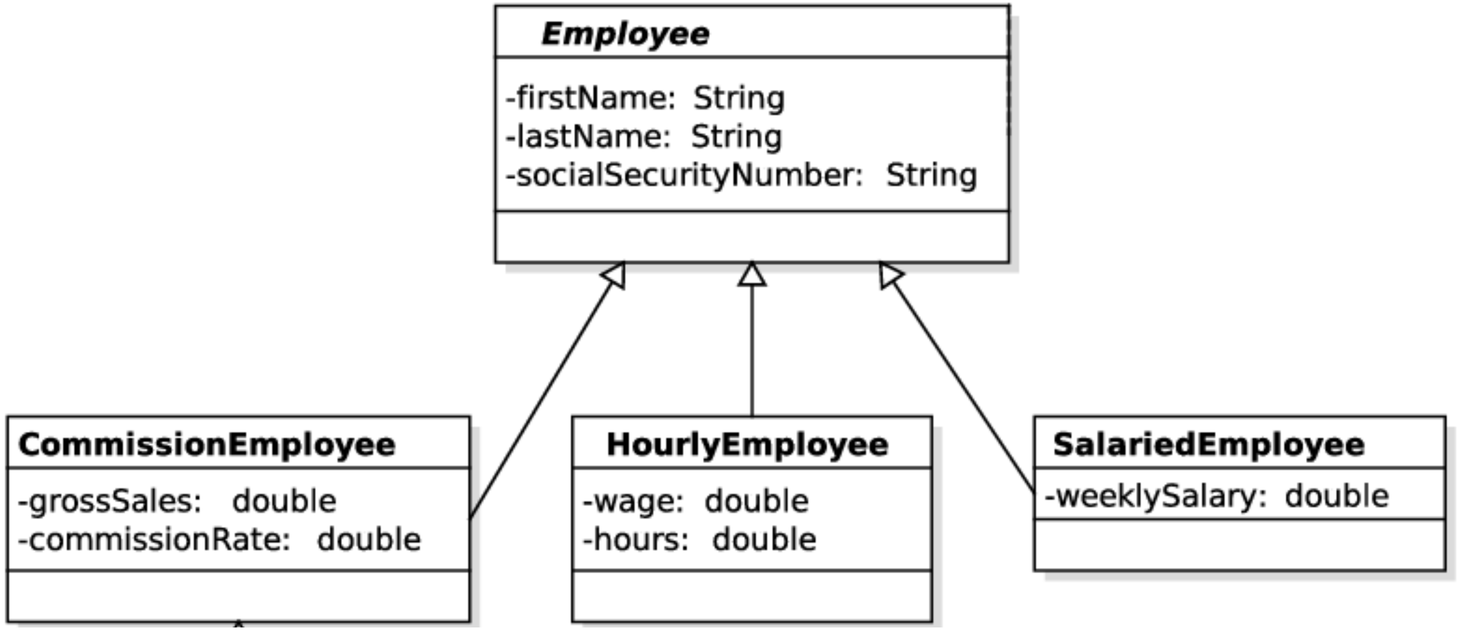


If Polygon is an interface, do we need to change our UML diagram?

ANOTHER EXAMPLE: ACCOUNTING SYSTEM

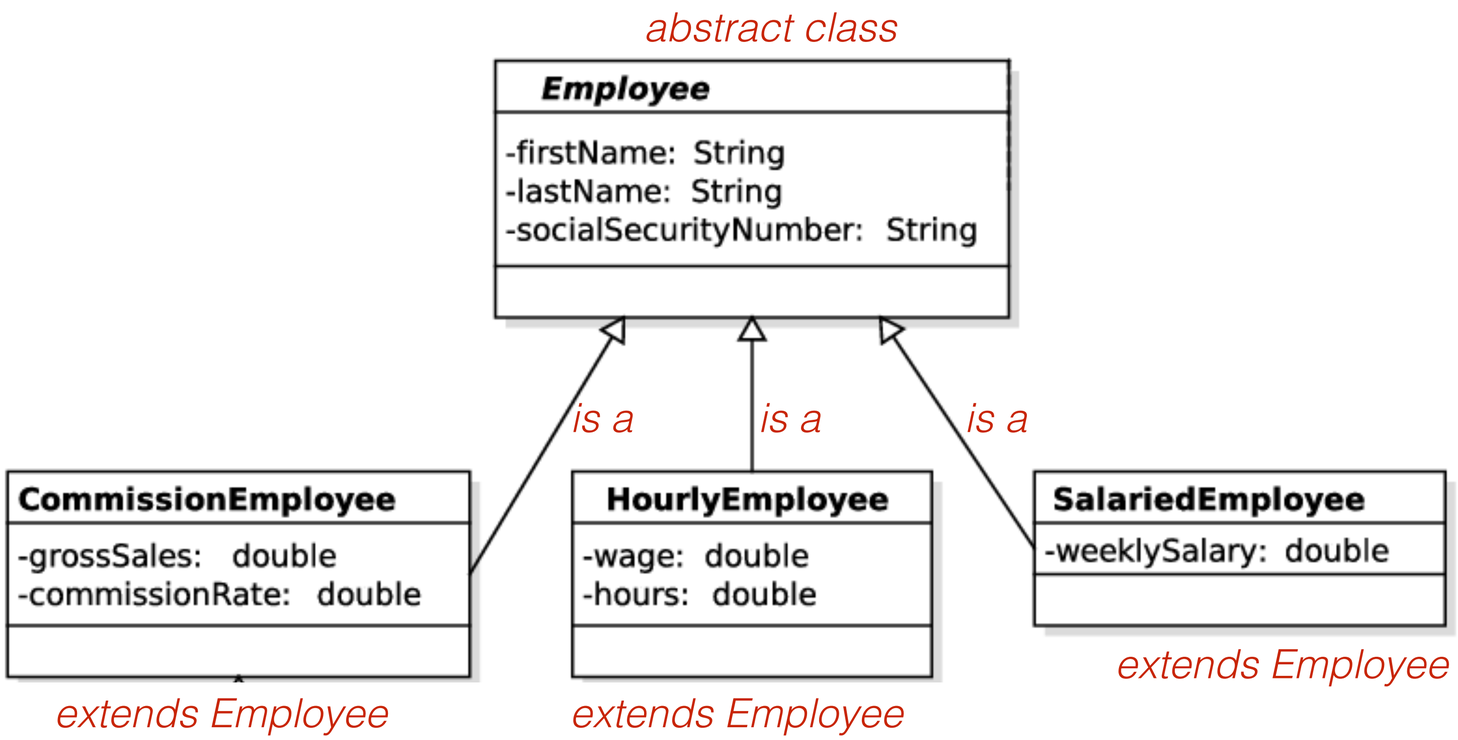
- Let's build an accounts payable system for a startup company!
- We'll need employees, and employees will need to be paid.
- Customers will order our products.
  - We'll create invoices to list product/part information, prices per part, and quantity of a part in the order.

ACCOUNTING SYSTEM



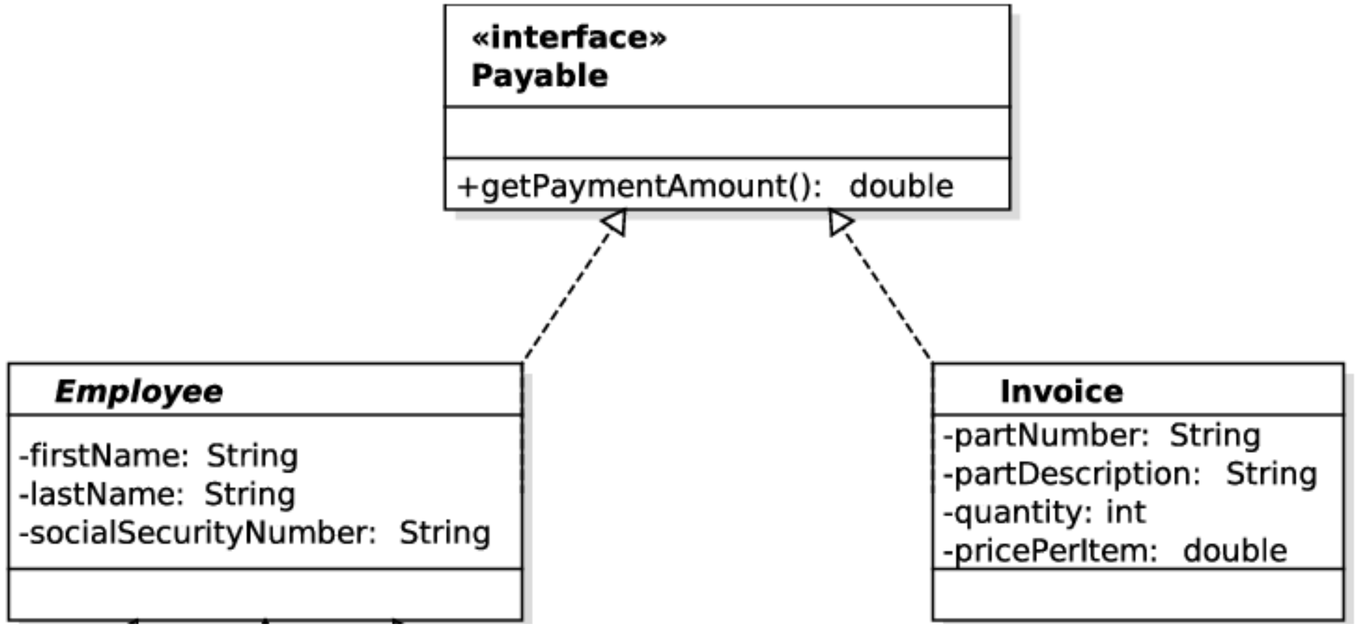
- This defines our commissioned, hourly, and salary employees.

ACCOUNTING SYSTEM



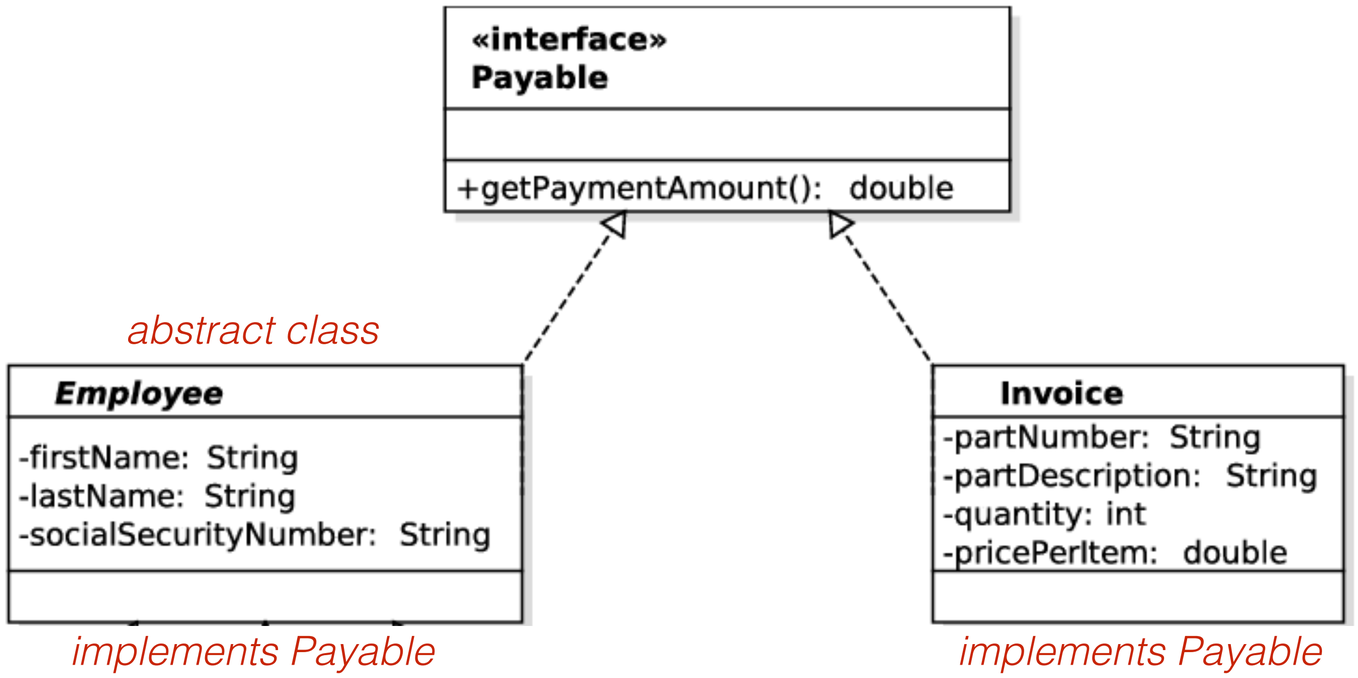
- This defines our commissioned, hourly, and salary employees.

ACCOUNTING SYSTEM

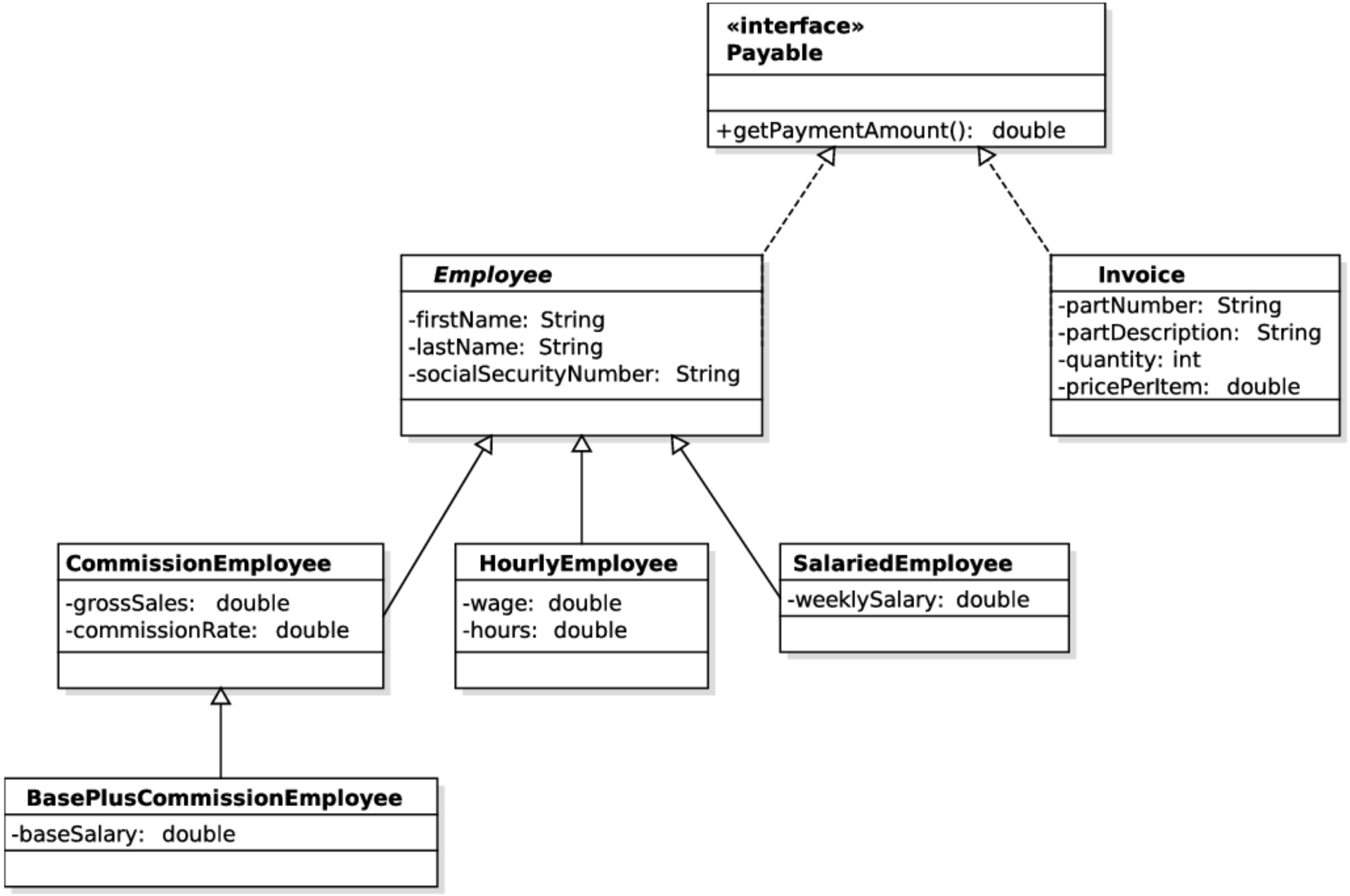


- Employees are payable.
- Invoices will also need to be in our system.

ACCOUNTING SYSTEM



- Employees are payable.
- Invoices will also need to be in our system.


«interface»
Payable
+getPaymentAmount(): double
Employee
-firstName: String
-lastName: String
-socialSecurityNumber: String
Invoice
-partNumber: String
-partDescription: String
-quantity: int
-pricePerItem: double
CommissionEmployee
-grossSales: double
-commissionRate: double
HourlyEmployee
-wage: double
-hours: double
SalariedEmployee
-weeklySalary: double
BasePlusCommissionEmployee
-baseSalary: double

ACCOUNTING SYSTEM

- Example code: Chapter 10

ACCOUNTING SYSTEM

- Creating an employee..

- Since `Employee` is an abstract class, this will cause an error:

```
Employee employee007 = new Employee("James", "Bond", "007-00-7007");
```

- A correct declaration & instantiation:

```
CommissionEmployee employee007 =
              new CommissionEmployee("James", "Bond", "007-00-7007");
```

GUI

Graphical User Interface

- Pronounced *“GOO-ee”*
- Intended to present a user-friendly mechanism for interacting with an application.
- GUIs are built from **GUI components**.
- An **event object** is created when the user interacts with it.
    - It’s ***dispatched*** to an **event handler** (*listener*).

MVC

Model-View-Controller

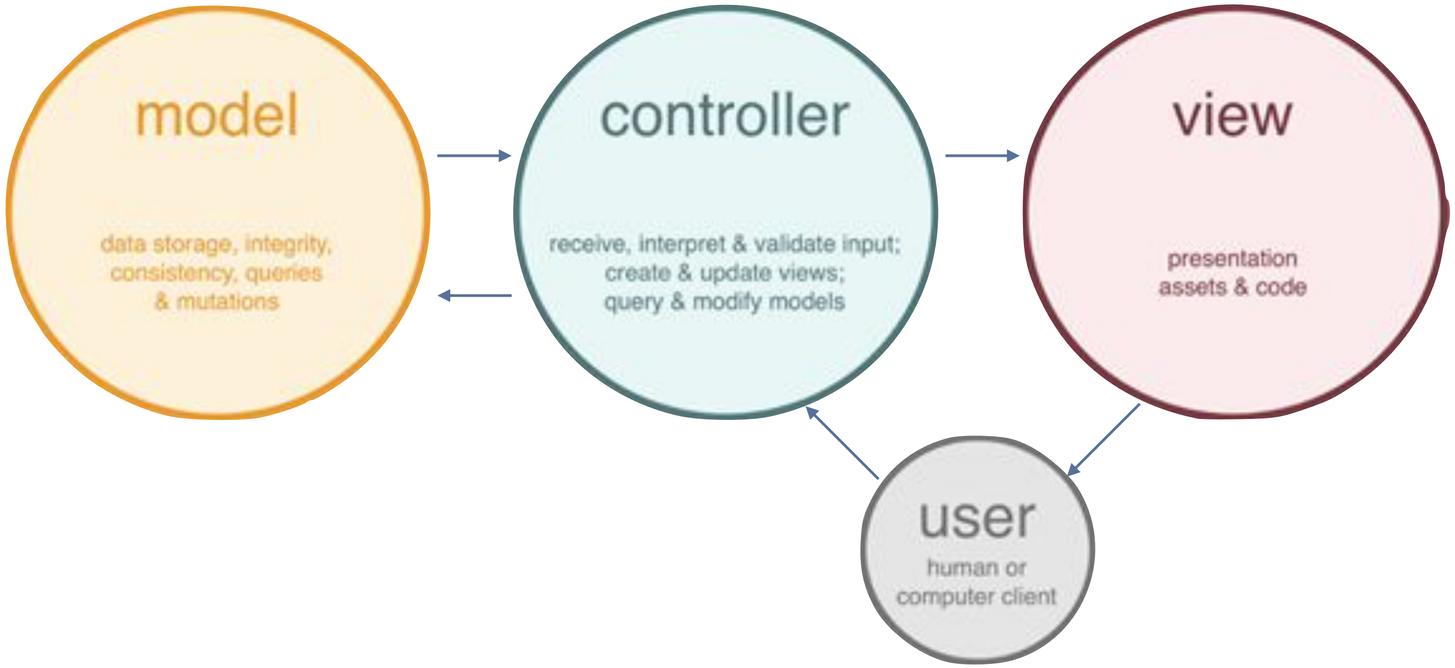
MVC is a design pattern which helps separate:

- the application logic from the user interface
- the control between the user interface and the application logic

This helps align code to the SOLID principles.

*we'll talk about these soon..

model-view-controller



model-view-controller

- In enterprise software, a model often serves as a software approximation of a real-world process.
- Implemented as 1 or more classes.
- The model represents the data and the rules that govern access to and updates of this data.

model-view-controller

- If the model data changes, the view must update its presentation as needed.
- Implemented as 1 or more classes.

- The view renders the contents of a model.
- The view specifies exactly how the model data should be presented.

model-view-controller

- Implemented as 1 or more classes.

- The controller translates the user's interactions with the view into actions that the model will perform.

GUI

- Graphical User Interface (GUI) is the *view* and *controller* of MVC design pattern, and provide a user-friendly mechanism for interacting with an application.

- GUIs are built from GUI **components**.

JavaFX & SceneBuilder

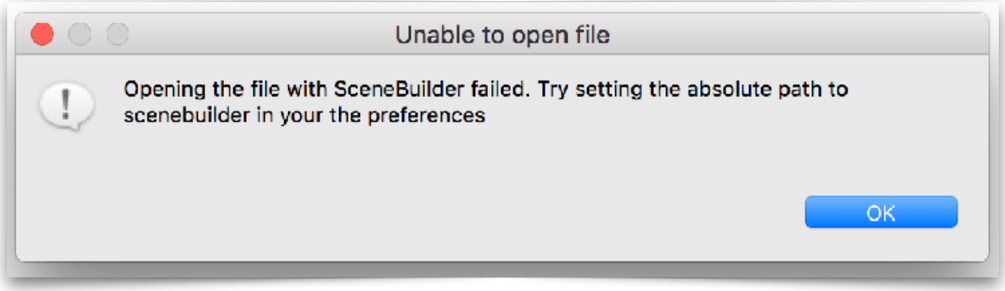
JavaFX

- **JavaFX** is the Java "*GUI, graphics, and multimedia API of the future*"*.

- **Scene Builder** is a graphical GUI builder to be used with JavaFX.
  - It's **WYSIWYG** (*"whizzy wig"*) - what you see is what you get.

SceneBuilder - install

SceneBuilder is already installed on the VDIs



With the default installation of Eclipse, you may receive the error above attempting to open a file with SceneBuilder. If so, follow these instructions:

1.Download SceneBuilder 2.0

1. http://www.oracle.com/technetwork/java/javase/downloads/javafxscenebuilder-info-2157684.html

2. In Eclipse, open Preferences > General > Editors > File Associations

3. Choose *.fxml and click “Add” next to Associated Editors.

4. Choose “external programs”, SceneBuilder.

5. Click “Apply and close”