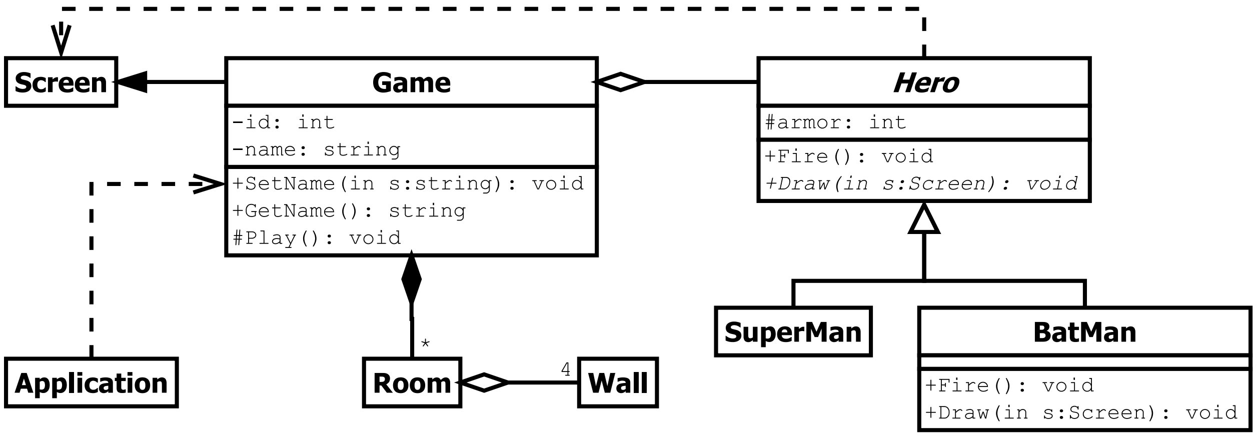
Cairo University
Faculty of Computers and Information
Computer Science Department

Programming-2 CS213
2018/2019

UML class diagrams

Dr. Amin Allam

The unified modeling language (UML) is a family of graphical notations that help describe software systems. One of these graphical notations is class diagrams.



The above example contains these classes: `Application`, `Screen`, `Game`, `Room`, `Wall`, `Hero`, `Superman`, and `Batman`. All of them are concrete classes (the class name is bold) except the abstract class `Hero` (the class name is bold and *oblique*).

Each class is composed of three parts: class name, member variables, and member functions. It is up to the designer to decide which information to include besides class name. Private members are preceded by `-`. Protected members are preceded by `#`. Public members are preceded by `+`. A pure virtual function name such as `Hero::Draw()` is *oblique*. The types of member variables and the return types of functions follow their names.

Classes `Superman` and `Batman` are inherited from class `Hero`. The inheritance relation is represented by a triangle.

An object of class `Game` is composed from objects of class `Room`. The composition relation is represented by a filled diamond. There can be arbitrary number of rooms in a game, which is represented by $\star$.

An object of class `Room` aggregates objects of class `Wall`. The aggregation relation is represented by a diamond. There are 4 walls in a room, which is represented by `4`. A wall object can be part of more than one room object (possibly 2 rooms). That is why it is an aggregation relation, not a composition relation.

An object of class `Game` aggregates one object of base class `Hero`. A `Hero` object can be used independently of `Game` and be aggregated in objects from other classes. That is why it is an aggregation relation, not a composition relation.

An object of class `Game` is associated to an object of class `Screen`. Class `Game` is not responsible of creating or destroying objects of class `Screen`, although they have a `Screen` object as a member variable. That is why it is association relation, not aggregation or composition. A `Game` object uses a `Screen` object, but not vice versa, which is indicated by the arrow direction.

The composition, aggregation, and association indicate that an object is a member variable of another object. Since they are represented in the diagram by arrows, there is no need to explicitly list them as member variables in the owner classes. Other relatively temporary dependencies are represented by dashed arrows, such as when an object of class `Hero` uses an object of class `Screen` to draw itself, and when the class `Application` creates and runs an object of class `Game`.