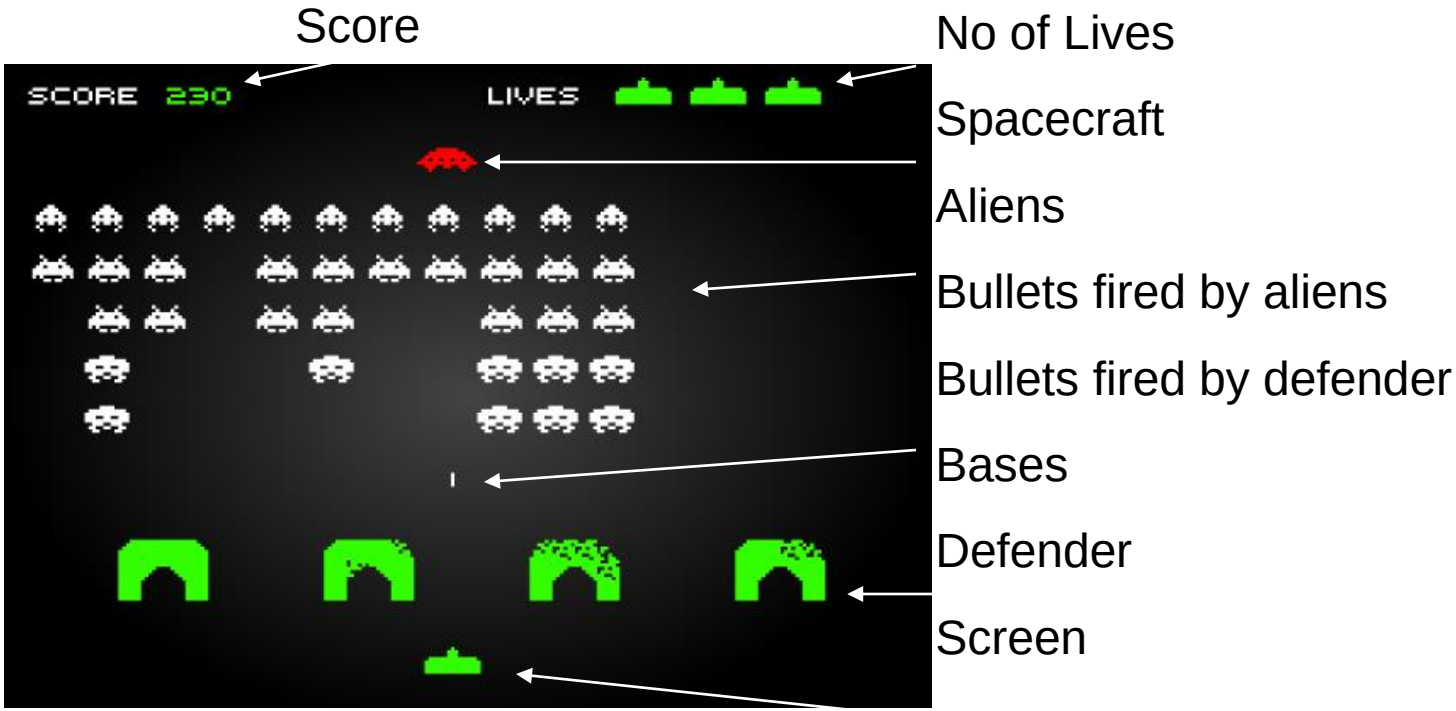
Class Diagrams

Space Invaders

A little simplified!

Looking for: Game Entities\Objects\Tokens:



I have simply listed everything I see on the screen (and the screen).

No of Lives - just a value

Score – just a value

Spacecraft – an object

Aliens – an object

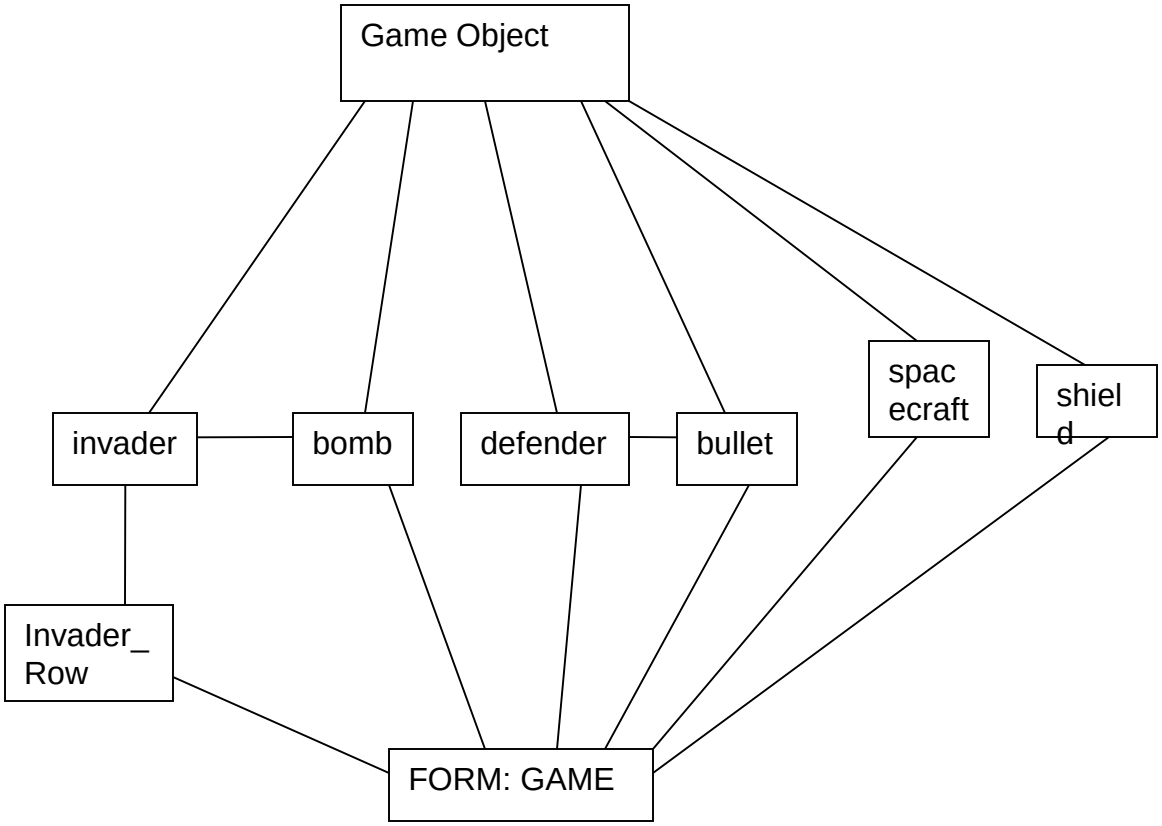
Bullets fired by aliens – an object

Bullets fired by defender – an object

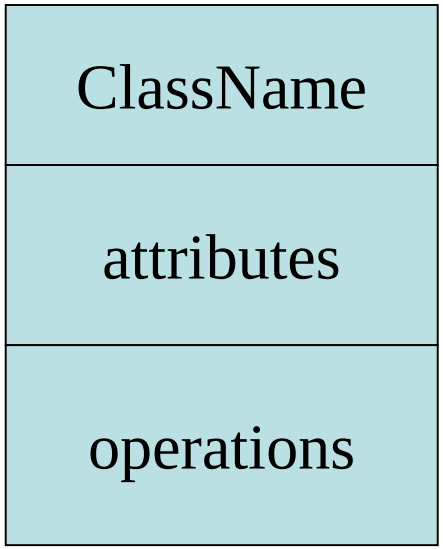
Bases – an object

Defender – an object

Screen ? – already a defined environmental object


Game Object
invader
bomb
defender
bullet
spac
ecraft
shiel
d
Invader_
Row
FORM: GAME

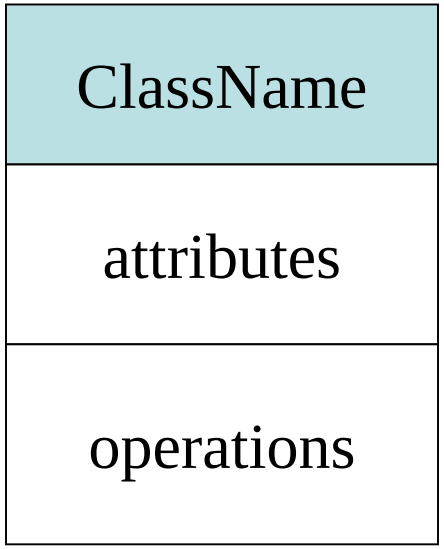
Classes



A *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics.

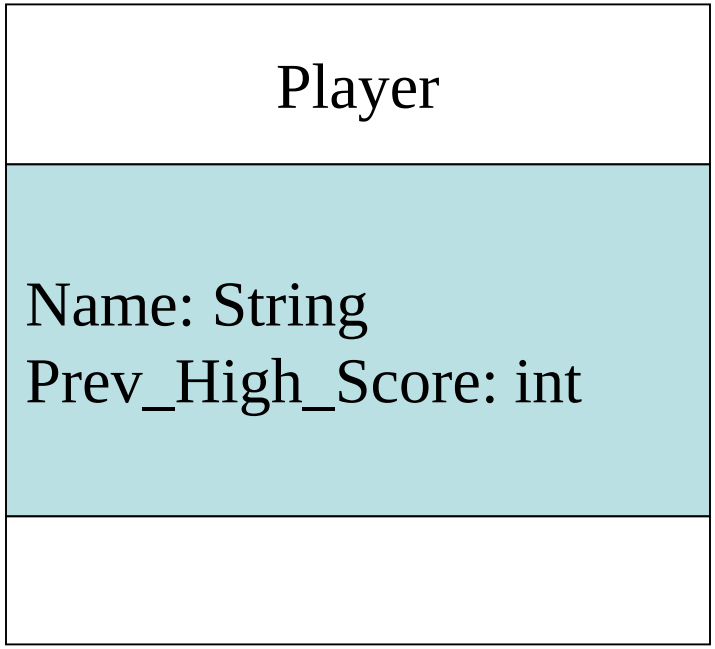
Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations in separate, designated compartments.

Class Names



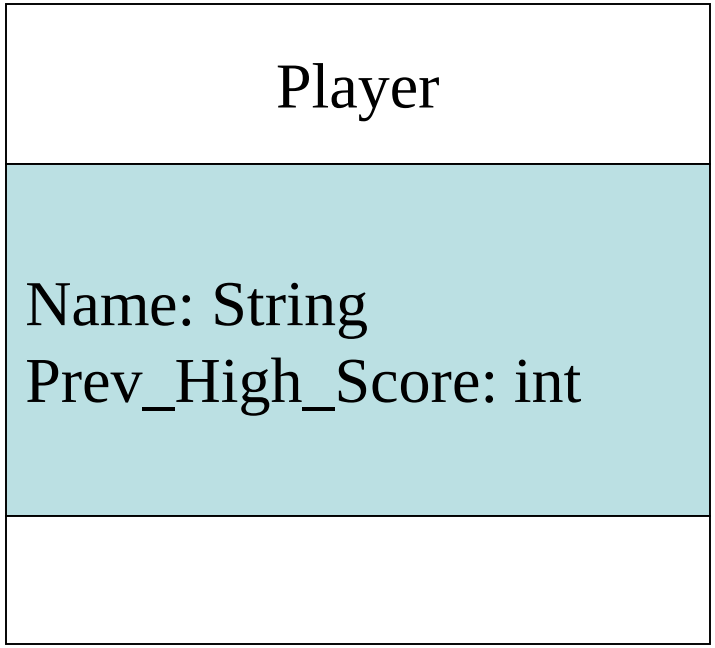
The name of the class is the only required tag in the graphical representation of a class. It always appears in the top-most compartment.

Class Attributes



An *attribute* is a named property of a
class that describes the object being modeled.
In the class diagram, attributes appear in
the second compartment just below the
name-compartment.

Class Attributes (Cont'd)



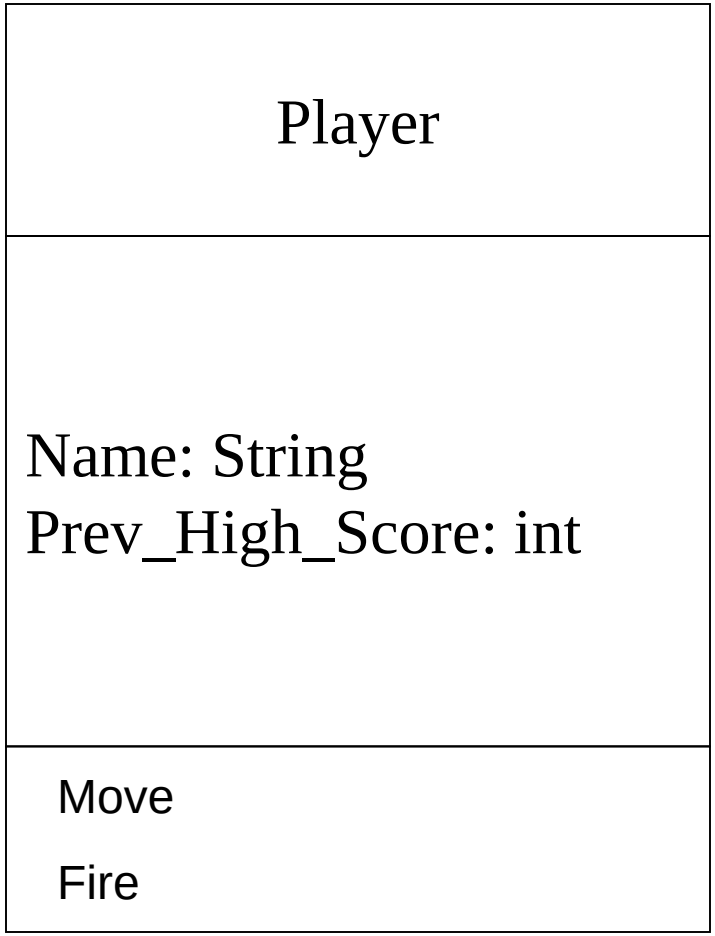
Attributes are usually listed in the form:

attributeName : Type

A *derived* attribute is one that can be computed from other attributes, but doesn't actually exist. For example, a Person's age can be computed from his birth date. A derived attribute is designated by a preceding '/' as in:

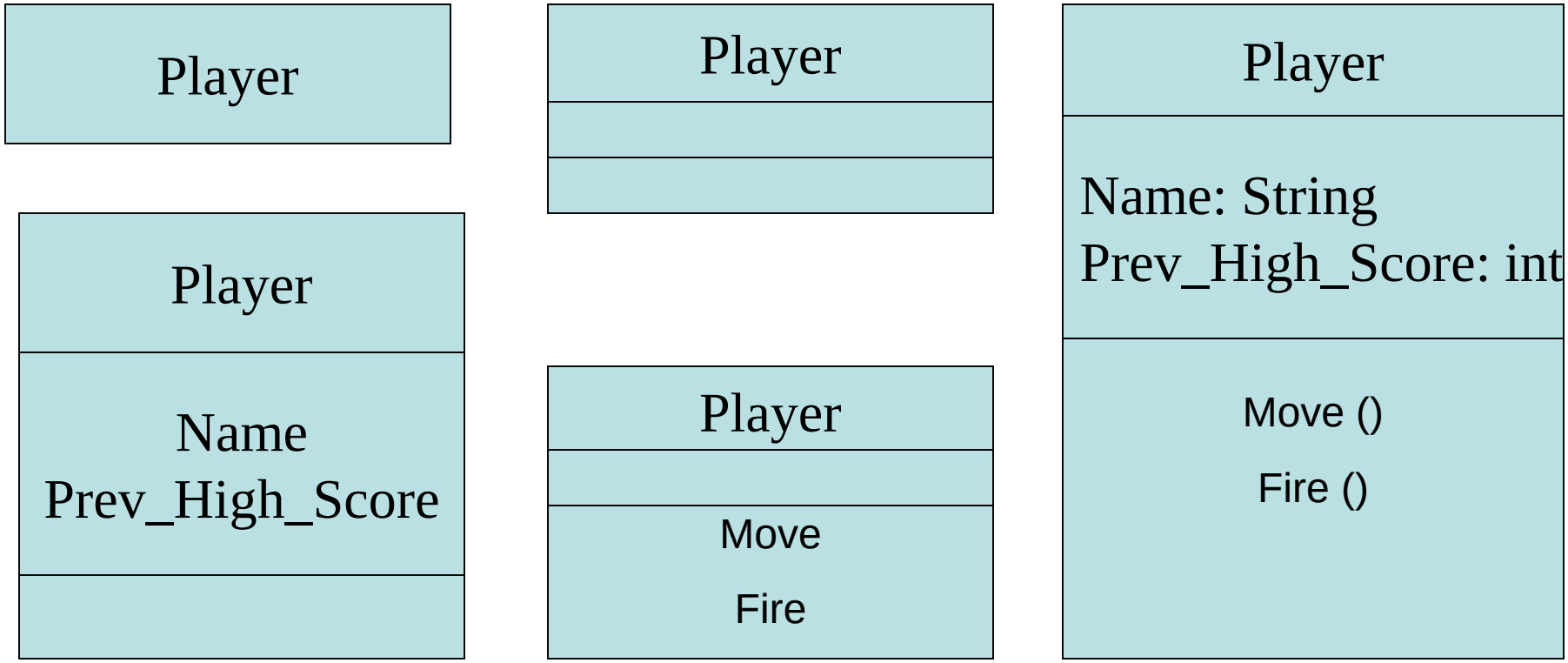
/ age : Date

Class Operations



Operations describe the class behavior
and appear in the third compartment.

Depicting Classes



When drawing a class, you needn't show attributes and operation in every diagram.

Class Operations (Cont'd)

Player
Move() Fire()

You can specify an operation by stating its signature: listing the name, type, and default value of all parameters, and, in the case of functions, a return type.

Relationships

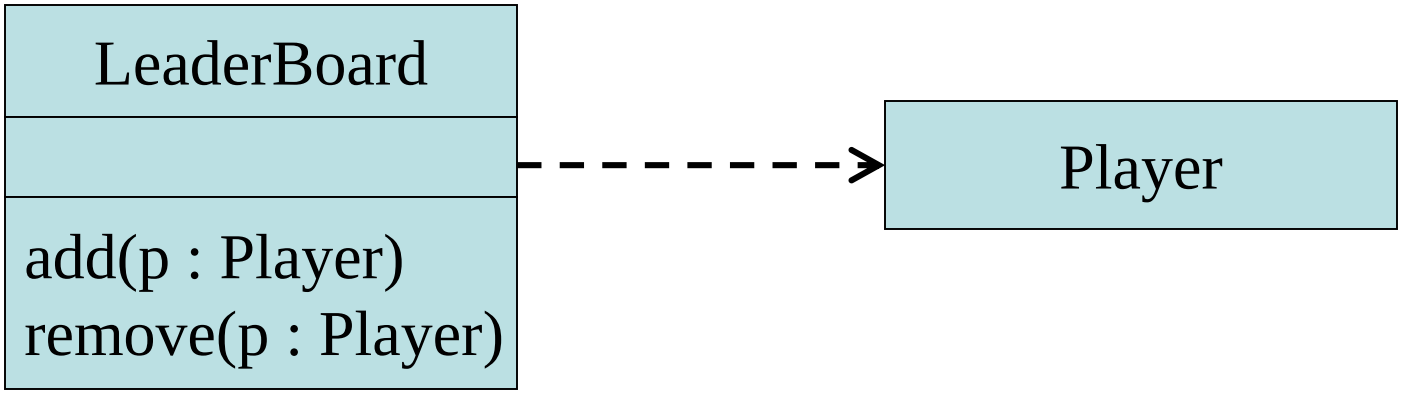
In UML, object interconnections (logical or physical), are modeled as relationships.

There are three kinds of relationships in UML:

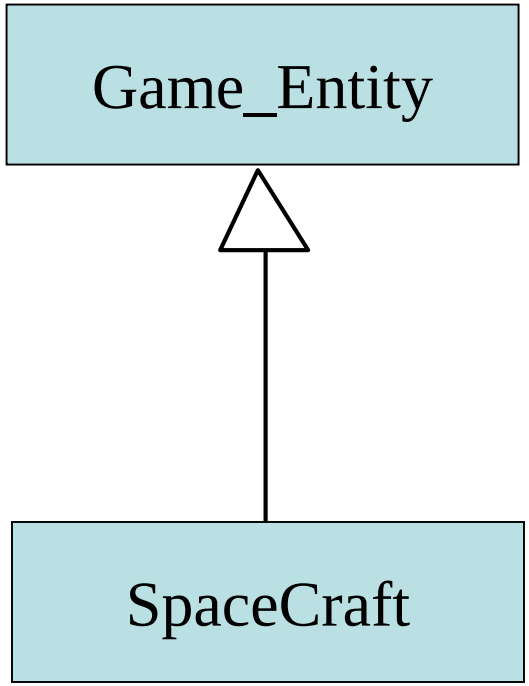
- dependencies
- generalizations
- associations

Dependency Relationships

A *dependency* indicates a semantic relationship between two or more elements. The dependency from *LeaderBoard* to *Player* exists because *Player* is used in both the **add** and **remove** operations of *LeaderBoard*.



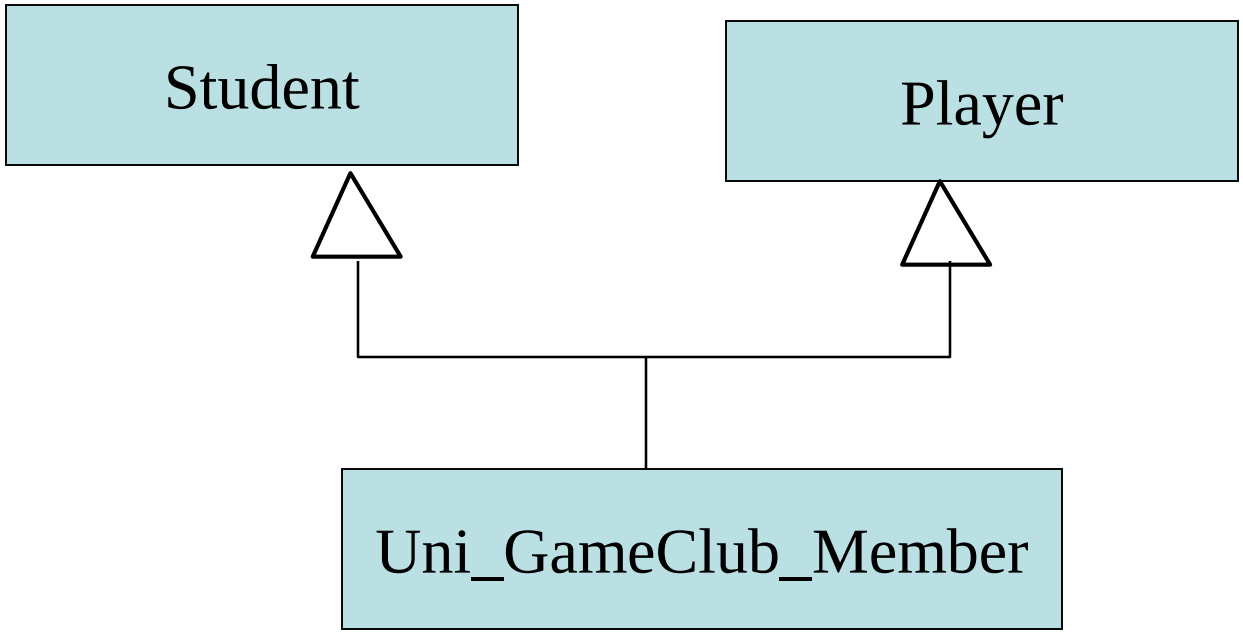
Generalization Relationships



A *generalization* connects a subclass to its superclass. It denotes an inheritance of attributes and behavior from the superclass to the subclass and indicates a specialization in the subclass of the more general superclass.

Generalization Relationships (Cont'd)

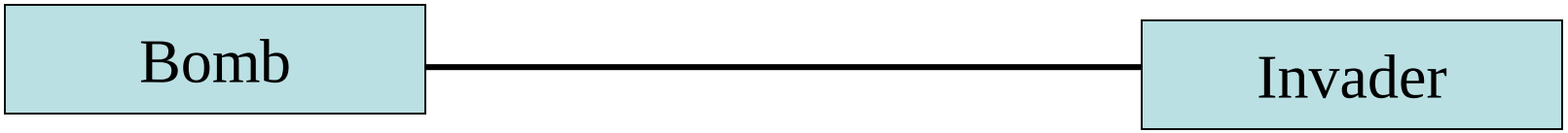
UML permits a class to inherit from multiple superclasses, although some programming languages (*e.g.*, Java) do not permit multiple inheritance.



Association Relationships

If two classes in a model need to communicate with each other, there must be link between them.

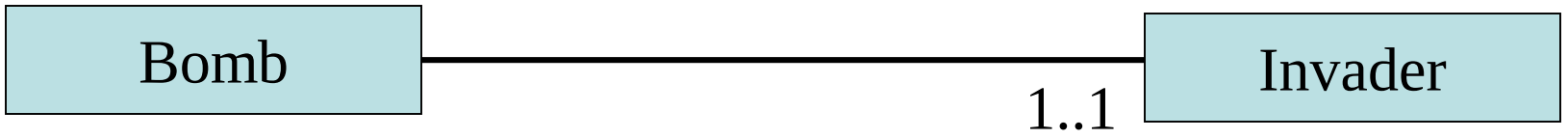
An *association* denotes that link.



Association Relationships (Cont'd)

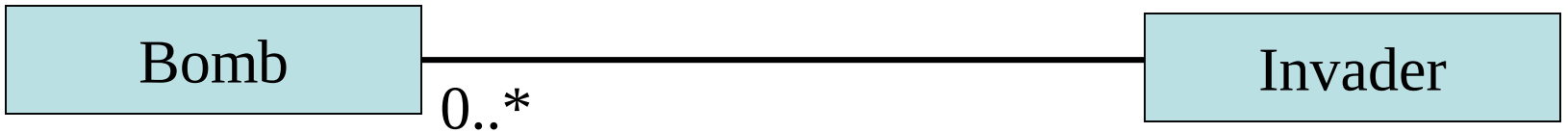
We can indicate the *multiplicity* of an association by adding *multiplicity adornments* to the line denoting the association.

The example indicates that a *bomb is dropped by one invader*



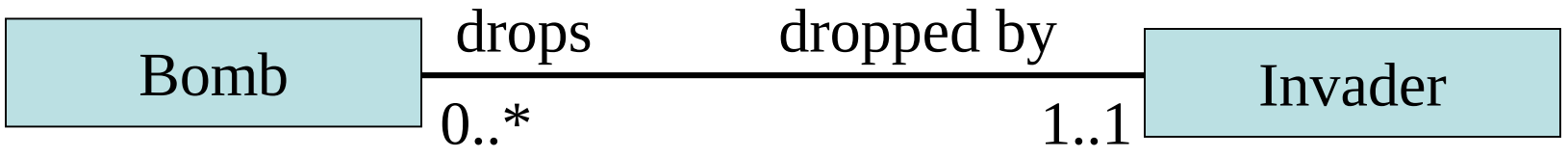
Association Relationships (Cont'd)

The example indicates that every *Invader* will drop zero to many bombs.



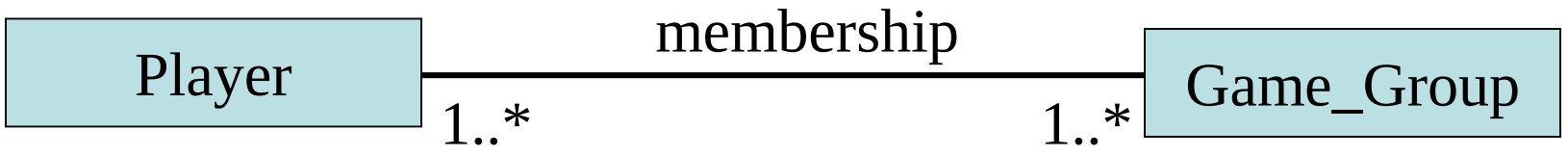
Association Relationships (Cont'd)

We can also indicate the behavior of an object in an association (*i.e.*, the *role* of an object) using *rolenames*.



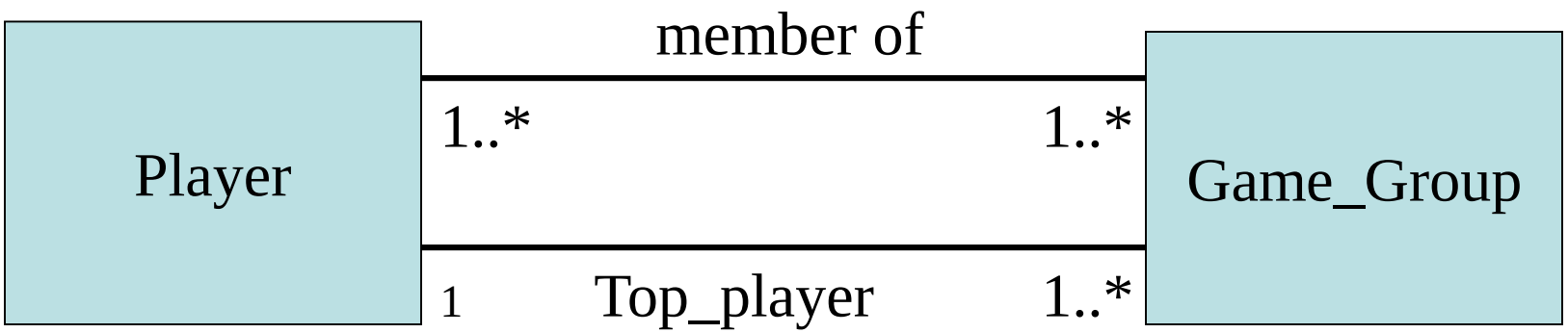
Association Relationships (Cont'd)

We can also name the association.



Association Relationships (Cont'd)

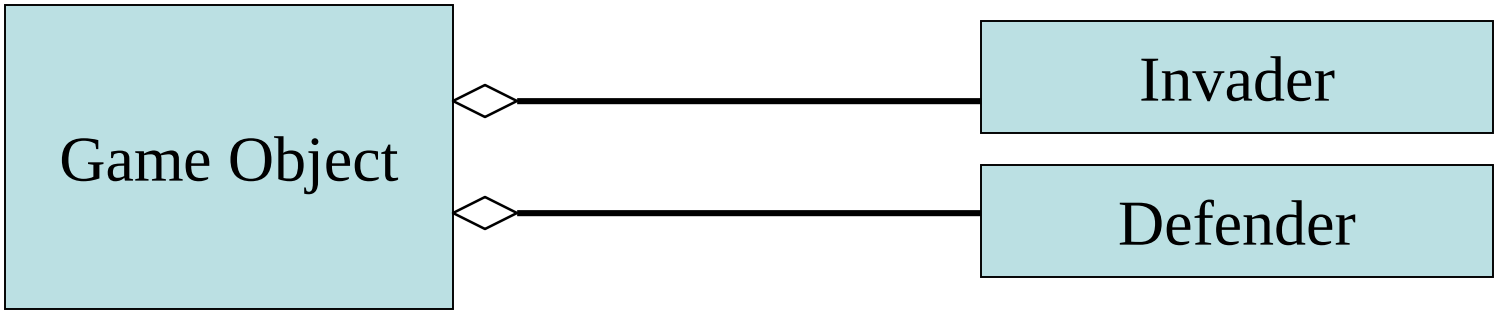
We can specify dual associations.



Association Relationships (Cont'd)

We can model objects that contain other objects by way of special associations called *aggregations* and *compositions*.

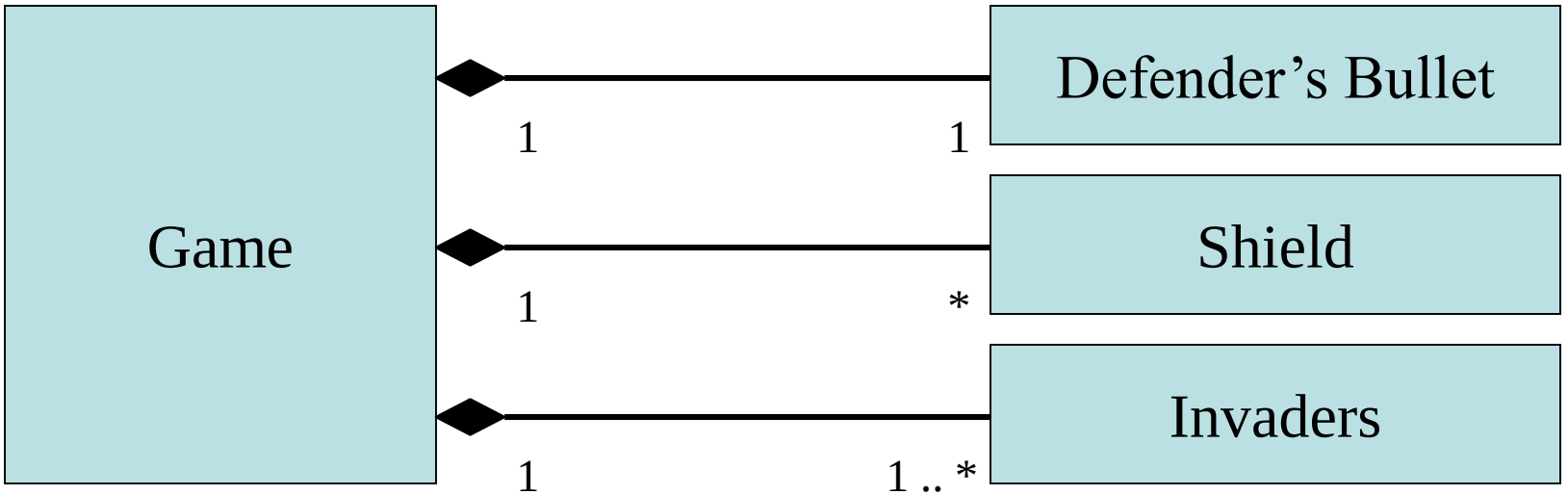
An *aggregation* specifies a whole-part relationship between an aggregate (a whole) and a constituent part, where the part can exist independently from the aggregate. Aggregations are denoted by a hollow-diamond adornment on the association.



And so on….

Association Relationships (Cont'd)

A *composition* indicates a strong ownership and coincident lifetime of parts by the whole (*i.e.*, they live and die as a whole). Compositions are denoted by a filled-diamond adornment on the association.



And so on….

Space invaders classes

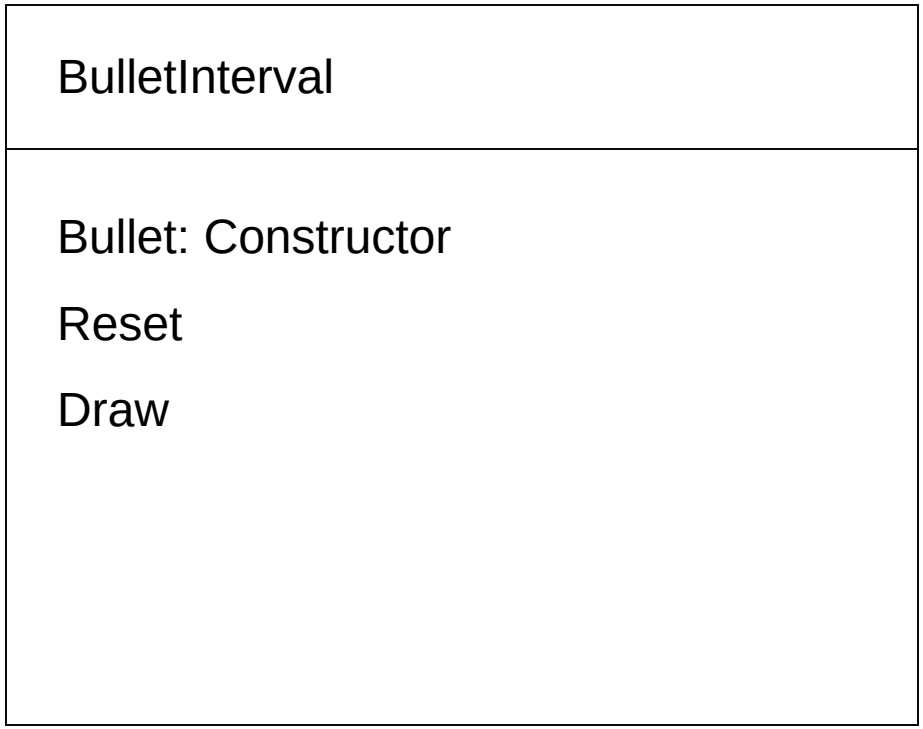
GameObject

Image,Position, ImageBounds,Moving_Bounds
GameObject: Constructor GetWidth GetHeight GetImage UpdatePosition Draw

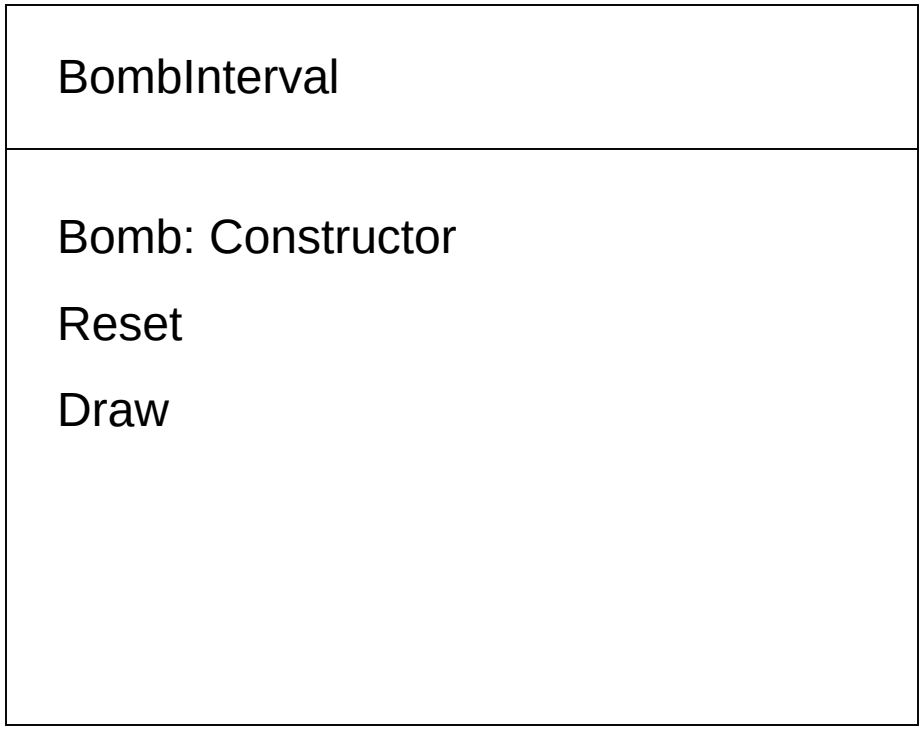
DEFENDER

MoveInterval, Died, beenhit
Defender: Constructor GetBulletStart MoveLeft MoveRight Draw

Bullet



Bomb



Invader

Otherimage, Bomb, Bomb_Interval, Active_Bomb, Died, Direction_Right,Interval
Invader: Constructor Draw Move Is BombColliding Get_Bomb_Bounds

InvaderRow

Invader[], lastposition, BombIntervalSpacing	
InvaderRow: Constructor Draw CollisionTest SetDirection Move Move_To_Initial ResetBombCounters	GetFirst GetLast Landed MoveDown

A more complete example for a C# Implementation

Form

No_Rows,NoTries, TimerCounter,Speed,Level,ActiveBullet,No_Defenders, Defender, GameGoing,Bullet, Invader_Rows[],Invader_Row,

Constructor

Initialise_Objects

Handle-Keys

Paint

Calc_LastPosition

Calc_FirstPosition

MoveInvaders

SetDirections

Calc_Score

BulletCollisions

BombCollisions

Test_Invasion

ResetBombCounters

Timer_Start

TokenGeneral

```
internal bool active = false;

    protected PointF location;
    protected Rectangle bounds;

    protected ImageAttributes ;
    protected Image sprite;

     abstract void Step (double elapsed);
     abstract void Render (Graphics g);

     Protected PointF Location{get{return location;}}
     Protected Rectangle Bounds {get{return bounds;}}
```

Defender

bool dead = false;

bool Dead { get { return dead; } }

Defender(PointF startLocation)

override void Render(Graphics g)

override void Step(double elapsed)

PointF GetBulletStartLocation()

Bullet

Bullet(PointF startLocation)

override void Render(Graphics g)

override void Step(double elapsed)

void Hit()

Alien

bool Dead = false;

Alien(PointF startLocation)

override void Render(Graphics g)

override void Step(double elapsed)

void Hit()

AlienGroup

private Alien[,] aliens;
private int leftMostAlien, rightMostAlien;

AlienGroup(int cols, int rows)

void CheckForCollision(Bullet bullet)

void Step(double elapsed)

void Render(Graphics graphics)

private void checkAlienDirection()

Mainform

Static Game game = new Game();

internal MainForm()

protected override void Dispose(bool disposing)

static void Main()

Game

```
private AlienGroup aliens;
private Image buffer;
private Graphics bufferGraphics;
private Graphics displayGraphics;
private Form form;
private Font font = new Font("Impact", 14);
private Font largeFont = new Font("Impact", 26);
private Brush fontBrush = Brushes.White;
private double renderElapsed = 0d;
private Defender defender;
private Bullet bullet;
```

Game

void Initialize(Form mainForm)

void GameLoop()

private void step(double elapsed)

private void render()

void OnKeyDown(object sender, KeyEventArgs e)

void startNewGame()

void detectCollision()

TokenGeneral

```
bool active = false;
protected PointF location;
protected Rectangle bounds;
protected Image sprite;
```

```
abstract void Step (double elapsed);
abstract void Render (Graphics g);

PointF Location{get{return location;}}
Rectangle Bounds {get{return bounds;}}
```

Global

```
static readonly Size FormSize  = new Size(800, 600);


static int Score = 0;
static bool GameOver = true;
static bool LevelFinished = false
static int CurrentLevel = 1;
static int PlayersRemaining = 3;
```

Global

```
 static Directions DefenderDirection = Directions.None;
 static float DefenderSpeed = 1f / 5f;

static readonly Size DefenderSize = new Size(40, 40);



 static Directions AlienDirection = Directions.Right;
 static float AlienSpeed =  1f / 20f;
static readonly Size AlienSize         = new Size(40, 40);
 static readonly Size AlienSeparation   = new Size(30, 20);

const int AliensRow = 10;
 const int AliensCol = 4;


 static readonly Size BulletSize        = new Size(10, 17);
 static bool bulletfiring = false;
```

Direction

enum Directions;