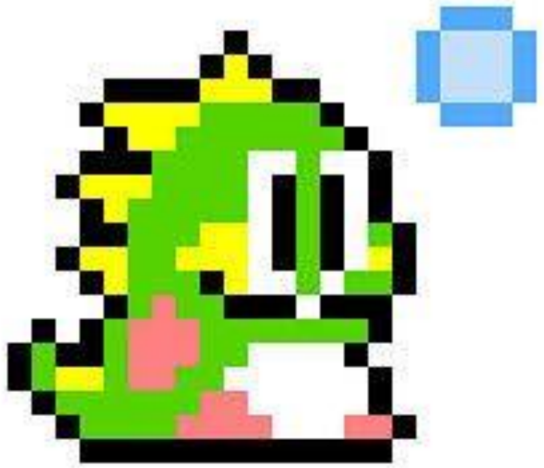
Game Programming

Lecture 4: User Interface, Inheritance

January 9, 2023

Paul Bonsma



Previous Weeks

- Previous weeks:
  - Getting started: creating a movable, animated sprite
  - Game objects, interaction, collisions
  - Level loading

→ You should be able to create *game play & levels* now

- If you have *huge levels*, and problems with *frame rate (performance)*:
  - Don’t add colliders if you don’t need them
  - For many tiles without colliders: check out the AddOns/SpriteBatch class
  - Last lecture is about optimization (among other things)
  - If you want, I can answer questions at the end of the lecture

This Week

- Creating a *user interface*
- Some advanced topics:
  - Events
  - More on inheritance

- If you’re already overwhelmed:
  - Don’t worry: not everything is necessary to pass this course
  - However you should learn it at some point, as an engineer!
  - You can revisit this later
  - I’ll make clear what’s essential and what’s optional

Lecture Outline

- Bug Hunting (fixing the Level Loading)
- Events
- User Interface (HUD, EasyDraw)
- More on Inheritance (protected, virtual, override / when and why)
- Conclusions

Bug Hunting

All the subtle things that can go wrong with level loading

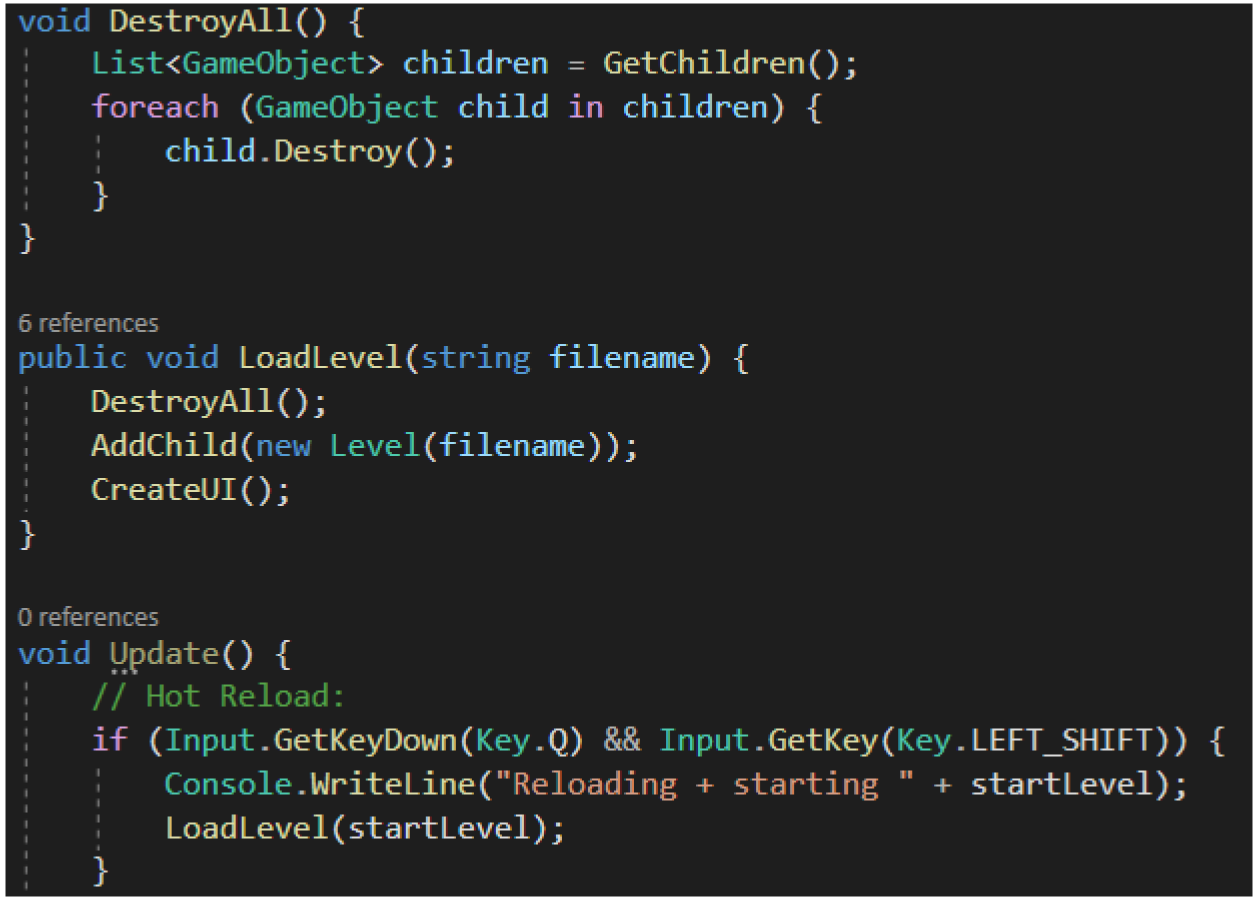
Warning

- Level loading / scene switching is often added last (this course / projects)
- …but this is one of the biggest causes of bugs / problems
- …and therefore, deadline stress / failures

- Let’s dig deeper now, while we’re still fresh and relaxed ☺

Last Week - MyGame

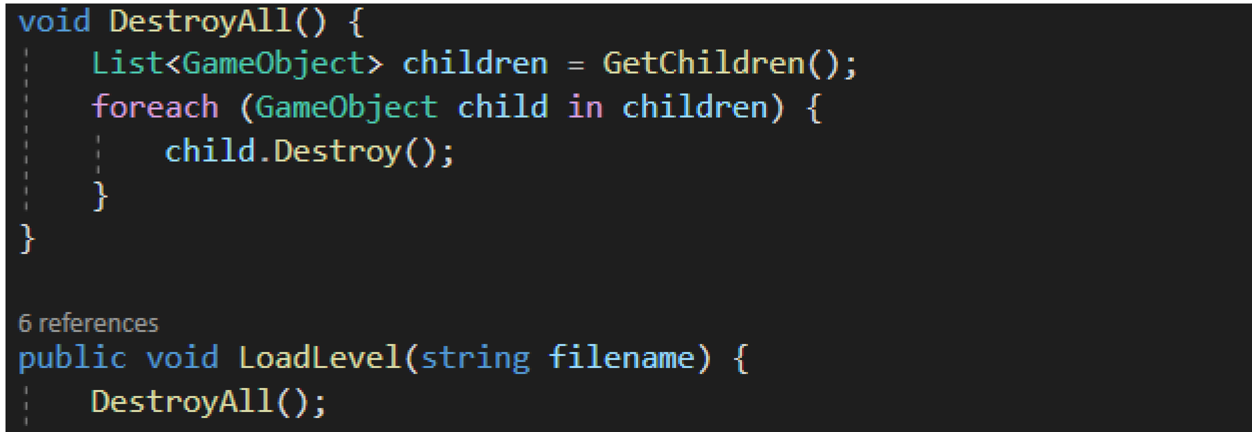
- Who has implemented (something like) this?
- …did you have any problems?
- What happens if LoadLevel is called from…
  - Update?
  - OnCollision?



```
void DestroyAll() {
    List<GameObject> children = GetChildren();
    foreach (GameObject child in children) {
        child.Destroy();
    }
}

6 references
public void LoadLevel(string filename) {
    DestroyAll();
    AddChild(new Level(filename));
    CreateUI();
}

0 references
void Update() {
    // Hot Reload:
    if (Input.GetKeyDown(Key.Q) && Input.GetKey(Key.LEFT_SHIFT)) {
        Console.WriteLine("Reloading + starting " + startLevel);
        LoadLevel(startLevel);
    }
```

Calling LoadLevel

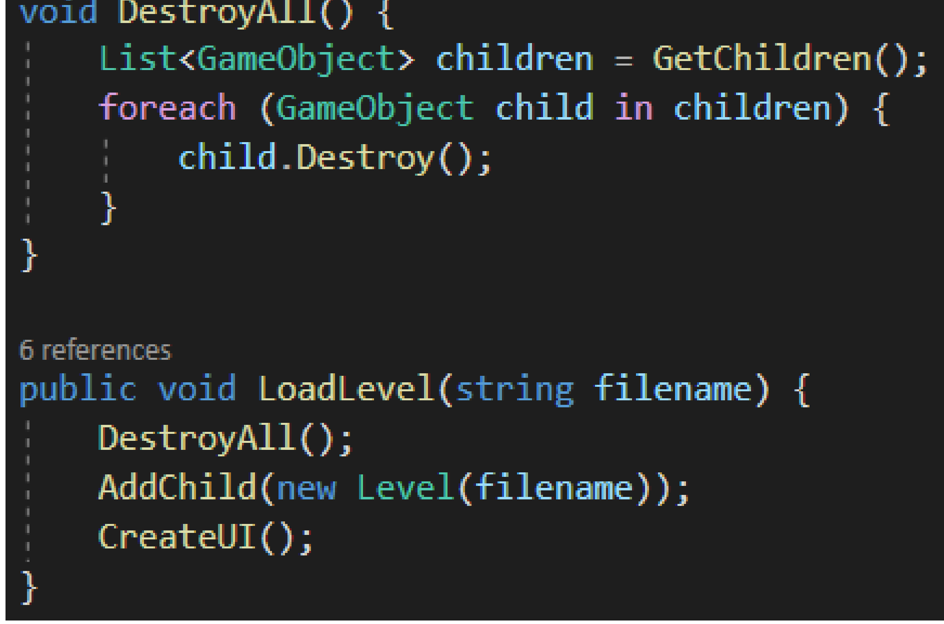
- Calling Destroy from *OnCollision*:
  - Gives Exception (can be turned off – but it's there for a good reason!)
- Suppose that during *Update* of object A (Level?), object B (Player?) is destroyed
  - Possible, but *B.Update() may still be called, after B is destroyed*!
  - This is a problem if *B.Update()* has something like *parent.AddChild(bullet)!*
  - (Why?)
- Solution: LateDestroy?



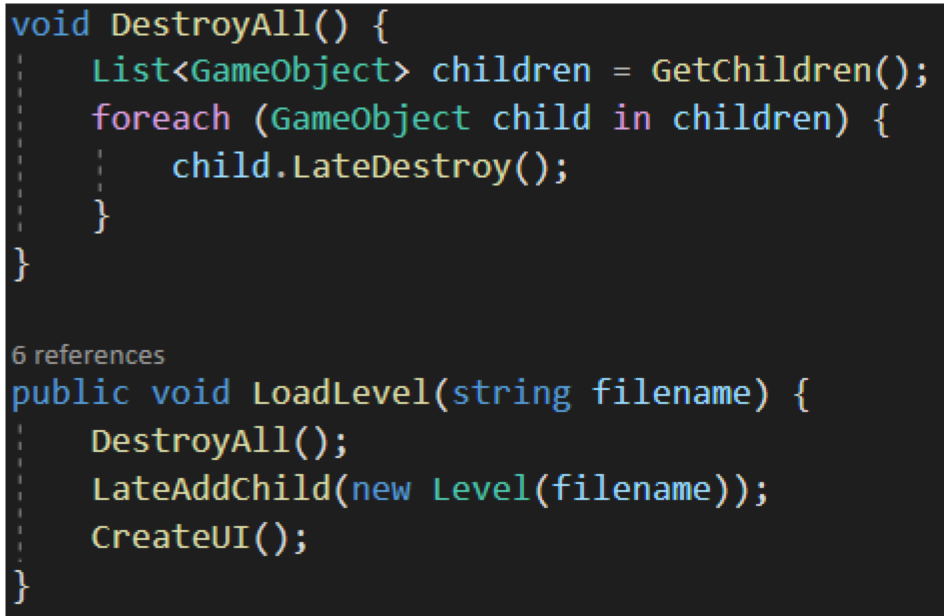
```
void DestroyAll() {
    List<GameObject> children = GetChildren();
    foreach (GameObject child in children) {
        child.Destroy();
    }
}

6 references
public void LoadLevel(string filename) {
    DestroyAll();
```

LateDestroy

- Solution: *LateDestroy* the previous level (and all children)?
- Subtle problem: for a short time, there are two levels (and two players). This might give problems when doing something like *player = FindObjectOfType<Player>();*
- Also: AddChild may also not be called from OnCollision…
- Solution: *LateAddChild*(level)?



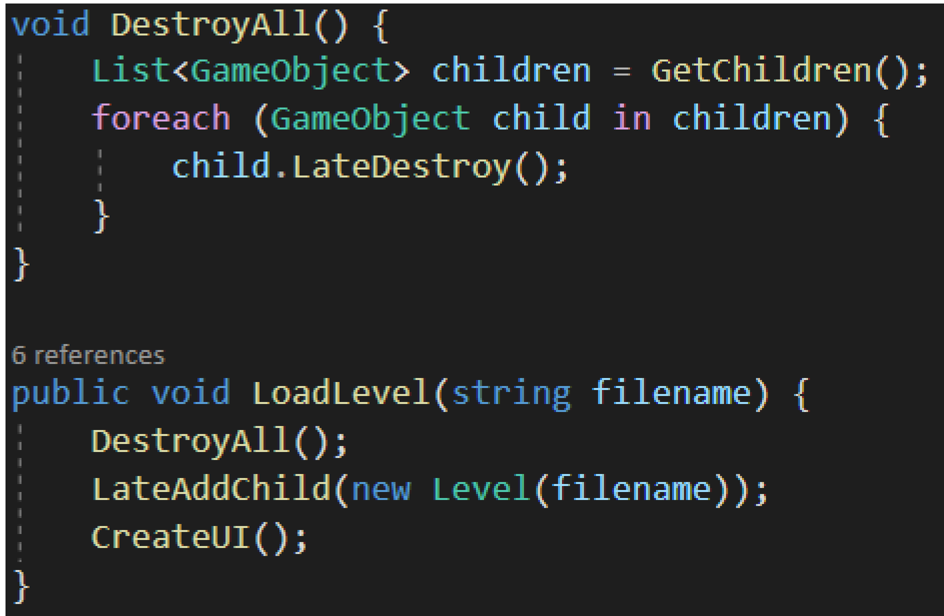
```
void DestroyAll() {
    List<GameObject> children = GetChildren();
    foreach (GameObject child in children) {
        child.Destroy();
    }
}

6 references
public void LoadLevel(string filename) {
    DestroyAll();
    AddChild(new Level(filename));
    CreateUI();
}
```



```
void DestroyAll() {
    List<GameObject> children = GetChildren();
    foreach (GameObject child in children) {
        child.LateDestroy();
    }
}

6 references
public void LoadLevel(string filename) {
    DestroyAll();
    LateAddChild(new Level(filename));
    CreateUI();
}
```

Another Problem

- What happens when LoadLevel is called *twice* in one frame? (e.g. player gets hit by two bullets at once!)
- *Bad:* the level file gets *loaded twice* in one frame!
- *Worse:* especially if we do LateAddChild, the first loaded level is not destroyed, and we will have *two active levels!*



```
void DestroyAll() {
    List<GameObject> children = GetChildren();
    foreach (GameObject child in children) {
        child.LateDestroy();
    }
}

6 references
public void LoadLevel(string filename) {
    DestroyAll();
    LateAddChild(new Level(filename));
    CreateUI();
}
```

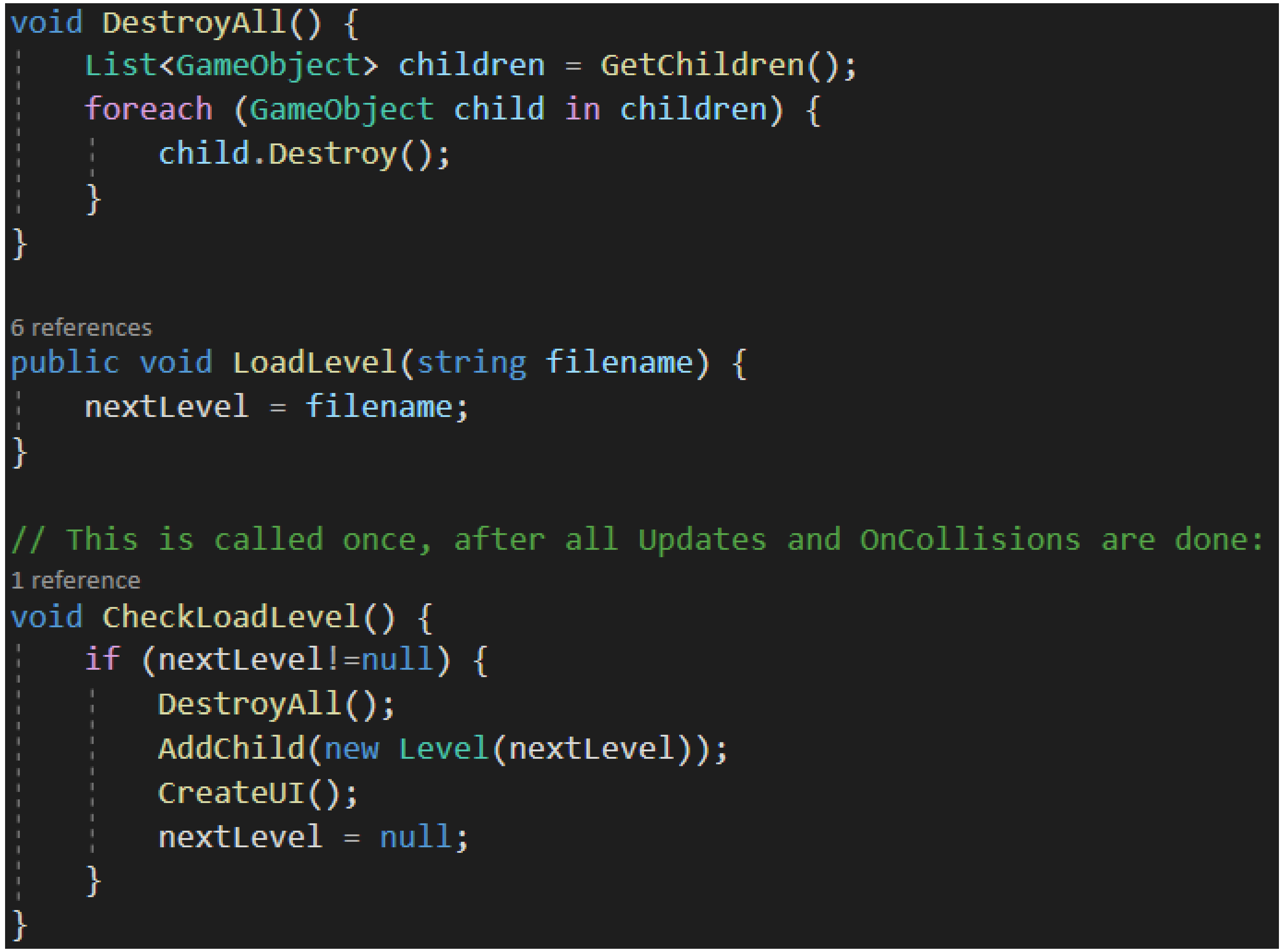
A Real Solution

- These bugs are really insidious, especially because they only occur very rarely (typically: only during an assessment), but then break everything

- We need a real solution!

- Core idea:
  - When *LoadLevel* is called, we store the *name* of the next level to be loaded
  - After *Update* and *OnCollision* are done, we destroy all old game objects, and then load that one new level
  - (If LoadLevel is called twice in one frame, only the second level is loaded)

A Real Solution



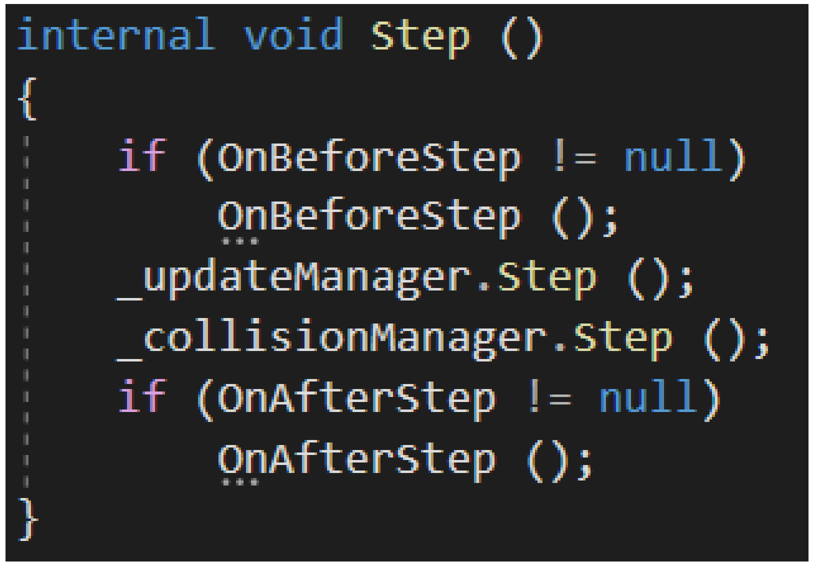
```
void DestroyAll() {
    List<GameObject> children = GetChildren();
    foreach (GameObject child in children) {
        child.Destroy();
    }
}

6 references
public void LoadLevel(string filename) {
    nextLevel = filename;
}

// This is called once, after all Updates and OnCollisions are done:
1 reference
void CheckLoadLevel() {
    if (nextLevel!=null) {
        DestroyAll();
        AddChild(new Level(nextLevel));
        CreateUI();
        nextLevel = null;
    }
}
```

$\leftarrow$ …but how do we make sure CheckLoadLevel is called after Update?

Events

Game Events

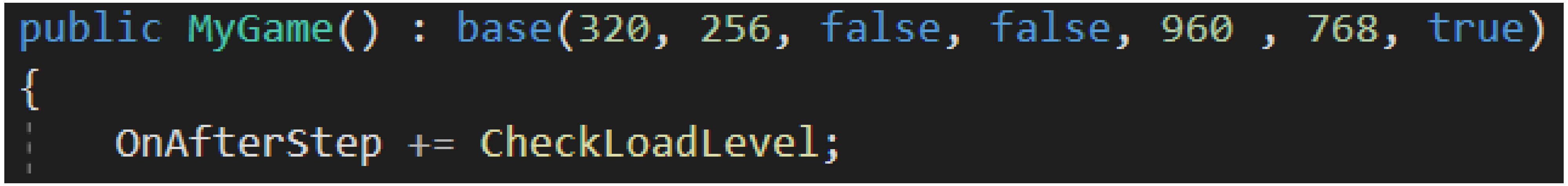
You can find this in the Game class:



```
internal void Step ()
{
    if (OnBeforeStep != null)
        OnBeforeStep ();
    _updateManager.Step ();
    _collisionManager.Step ();
    if (OnAfterStep != null)
        OnAfterStep ();
}
```

- The Game class contains *events*, that you can *subscribe* to:
  - *OnBeforeStep*: before Update & OnCollision
  - *OnAfterStep*: after Update & Collision (but before Render)
  - *OnAfterRender*: after Render

- You can subscribe as follows:



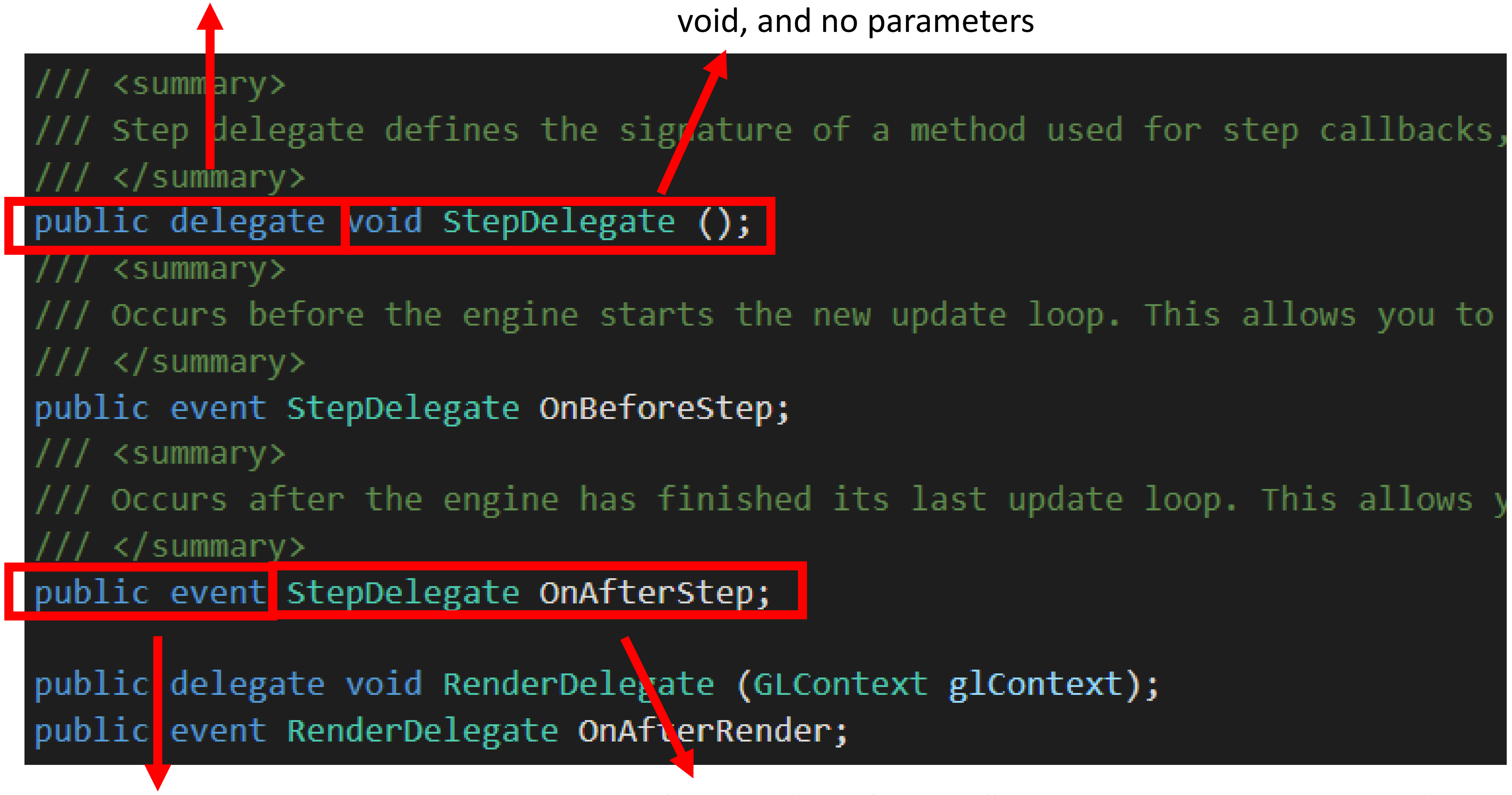
```
public MyGame() : base(320, 256, false, false, 960 , 768, true)
{
    OnAfterStep += CheckLoadLevel;
```

Events and Delegates

- An *event* can be seen as something that happens under certain conditions, and any number of methods can *listen* to it (=be *called* when it happens)
- You can *subscribe* to an event using += and *unsubscribe* using -=
- (Don't forget to unsubscribe, unless you can explain why it's not needed in your case!)
- You can only subscribe a method to an event if it has the correct *parameters*
    - In the case of CheckLoadLevel: no parameters
- These parameters are usually defined by a *delegate*
- The syntax and terminology is a bit cryptic – you don't need to know this to use it, but let's have a short look anyway (see the Game.cs class)

Here’s a public definition of a delegate

It has name “StepDelegate”, return type void, and no parameters



```
/// <summary>
/// Step delegate defines the signature of a method used for step callbacks,
/// </summary>
public delegate void StepDelegate ();
/// <summary>
/// Occurs before the engine starts the new update loop. This allows you to
/// </summary>
public event StepDelegate OnBeforeStep;
/// <summary>
/// Occurs after the engine has finished its last update loop. This allows y
/// </summary>
public event StepDelegate OnAfterStep;

public delegate void RenderDelegate (GLContext glContext);
public event RenderDelegate OnAfterRender;
```

Here’s a declaration of a public event

It’s called “OnAfterStep”, and only methods with the “StepDelegate” shape may subscribe to it

Events - Summary

- Using the built-in events like *OnAfterStep* can be really useful
- There are other useful events as well (e.g. TiledLoader: *OnObjectCreated*)
- You can define your *own events* as well – sometimes it’s very useful to keep your code clean and simple (e.g. your MyGame has an *OnLevelLoaded* event?)

- Using events is not needed to pass this course
- However, at some point, as a programmer, you’ll need to know this…

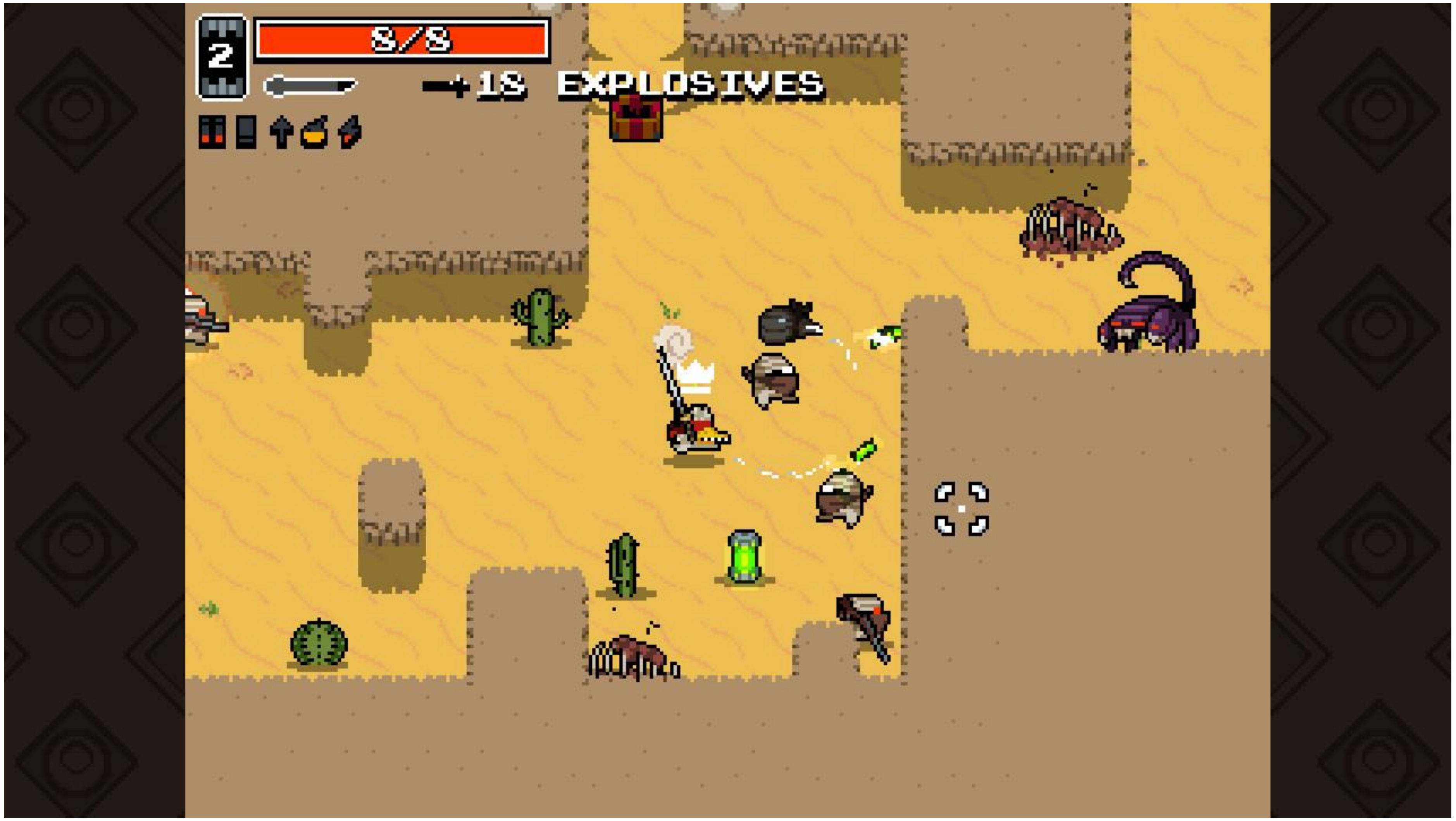
User Interfaces

User Interfaces / HUD

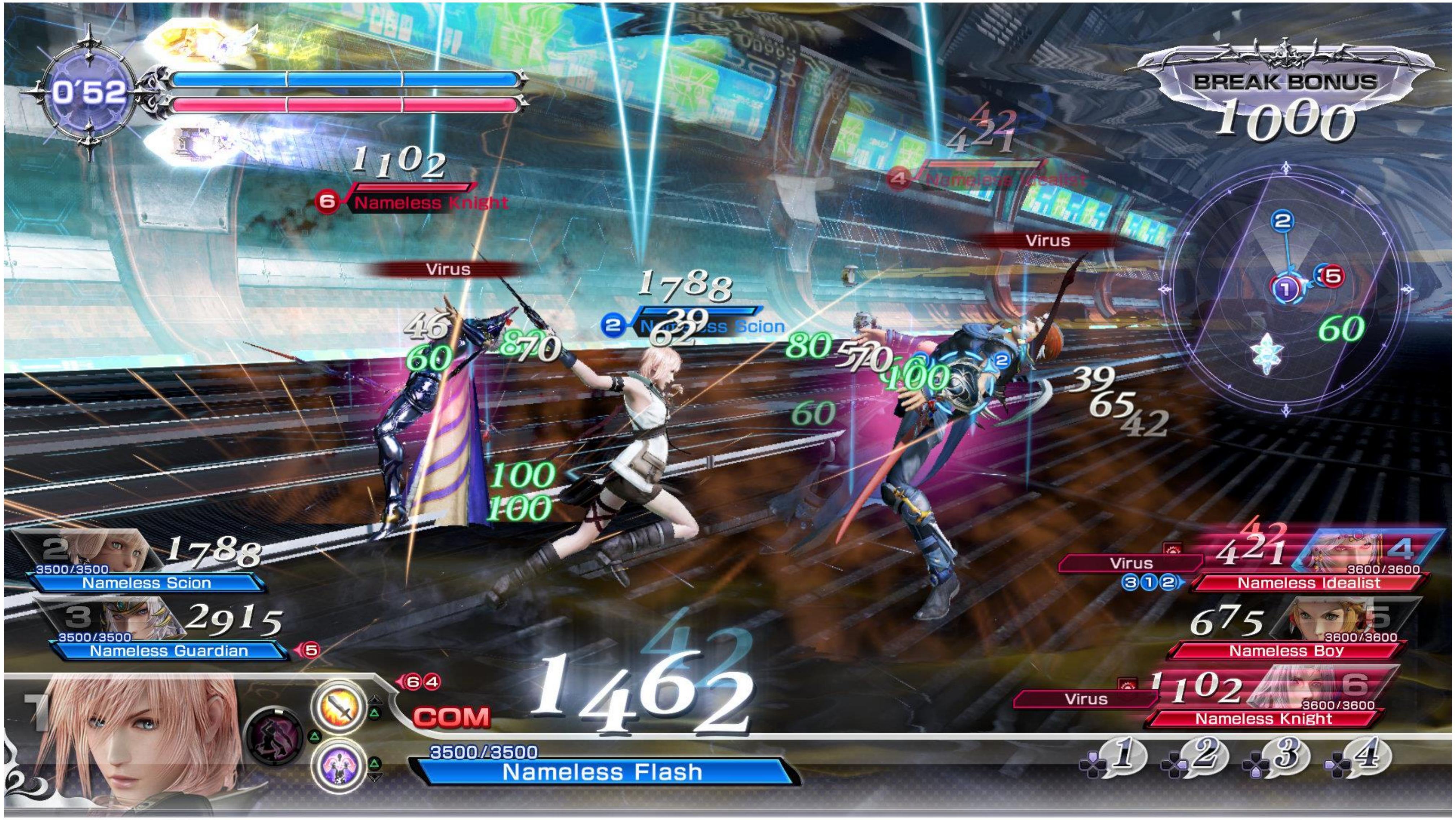
- Most games have a *user interface* to show relevant information to the player. Examples:
    - Health bar
    - Ammo count
    - Current objective
    - Score
    - Progress bar
    - Active weapon

- Usually an “overlay” (rendered over the game scene), with many text elements – it doesn’t scroll!
- Also called *HUD* (Heads Up Display)

Nuclear Throne

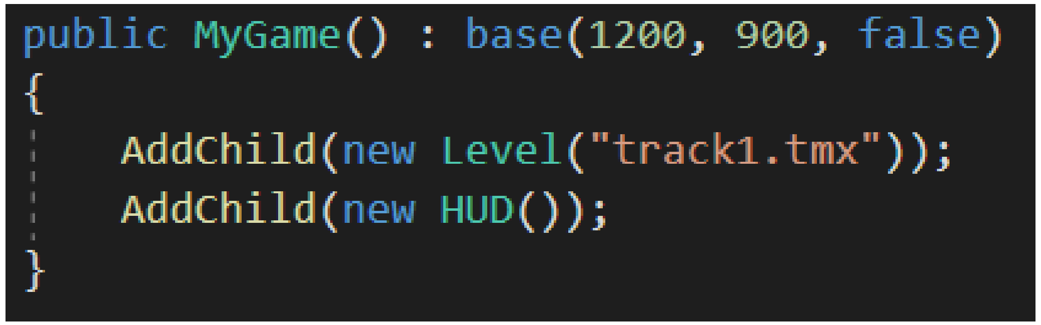


Too Much?!


0′52
BREAK BONUS
1000
1102
Nameless Knight
421
Nameless Idealist
Virus
Virus
1788
Nameless Scion
60
1788
3500/3500
Nameless Scion
2915
3500/3500
Nameless Guardian
421
3600/3600
Nameless Idealist
675
3600/3600
Nameless Boy
1102
3600/3600
Nameless Knight
COM
1462
3500/3500
Nameless Flash

Creating a HUD

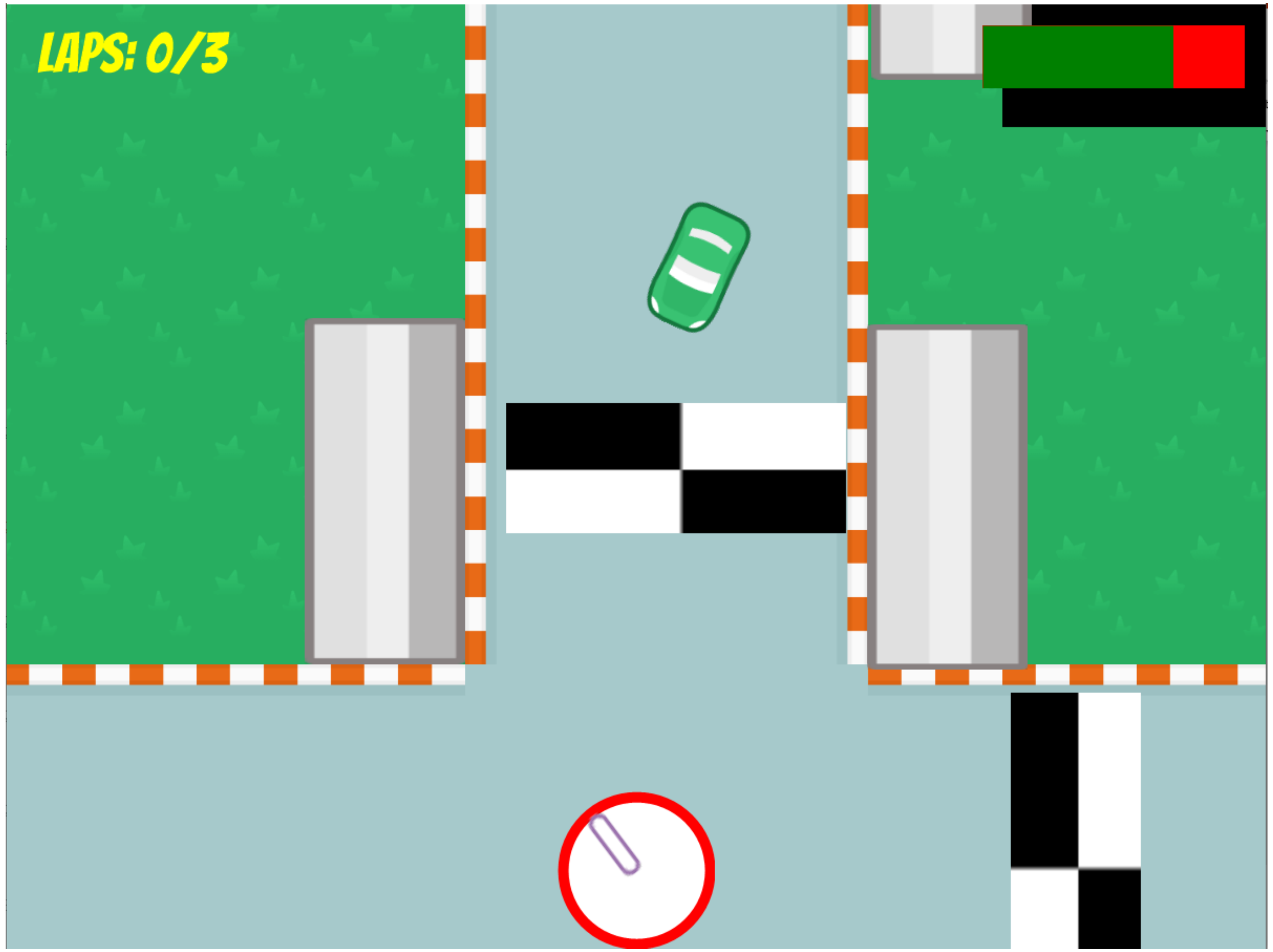
- Good *user interface design* is a separate subject, but let’s look at how we can create one



```
public MyGame() : base(1200, 900, false)
{
    AddChild(new Level("track1.tmx"));
    AddChild(new HUD());
}
```

- Setup:
  - Create a *HUD* class (a GameObject),
  - with public methods such as ShowHealth()
  - This class *encapsulates* all the drawing details
  - *MyGame* creates the HUD
  - If you want, you can design your HUD in *Tiled* as well, and load it using the TiledLoader! (Note: you only need to load the file once)

Today's Example Game

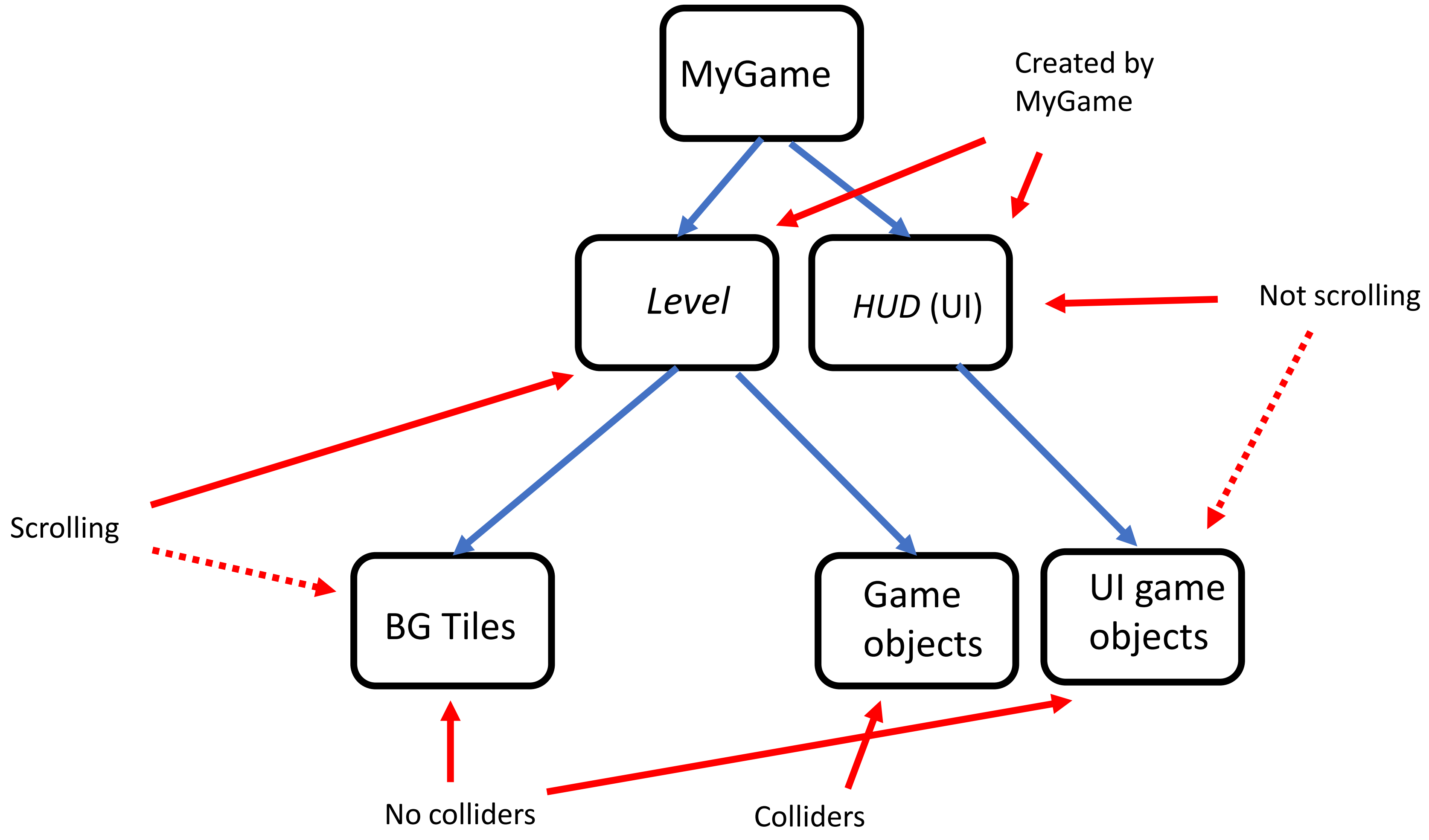


Sprites: www.kenney.nl
(racing pack)

HUD elements:
- Lap counter (text)
- Health bar
- Speedometer

All HUD elements are *EasyDraw* (Canvas) objects

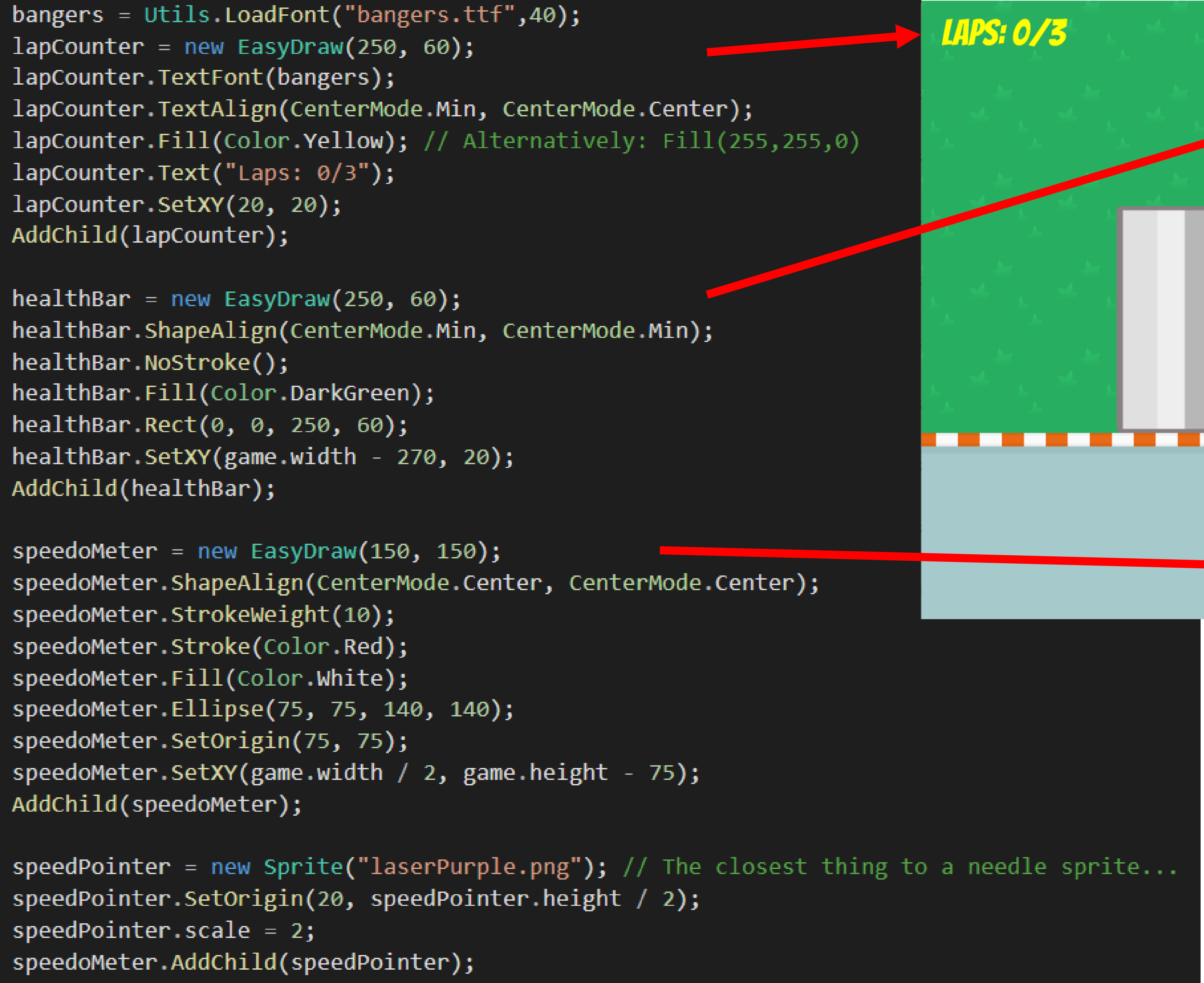
Hierarchy Overview



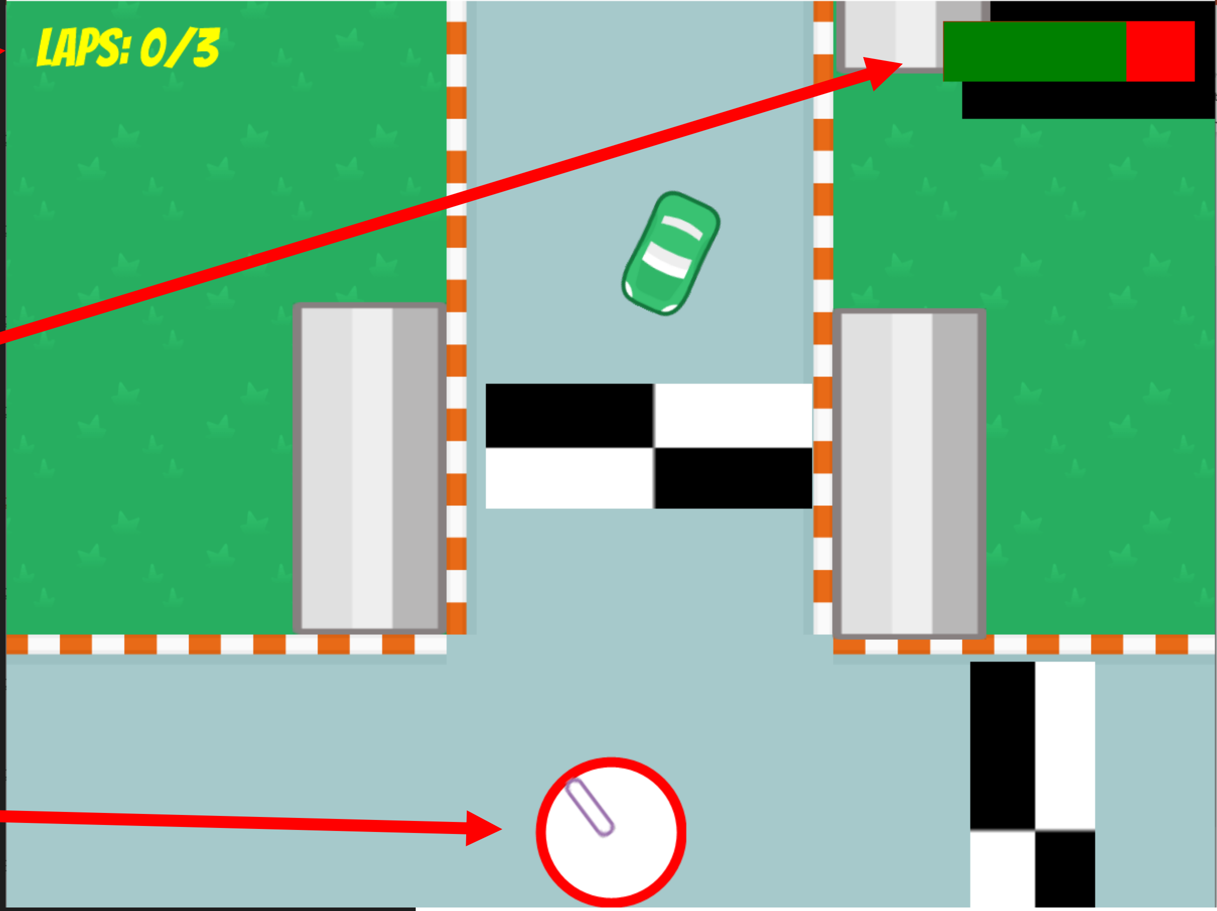
EasyDraw

- The *EasyDraw* class inherits from *Canvas*, which inherits from *Sprite*
- It can be used to draw *text* and *shapes* (lines, circles, rectangles, etc.)
- The interface is very similar to Processing:
  - Set the drawing *state* (outline + fill color, line width, font, etc.), and then
  - *Draw* text or shapes using those settings

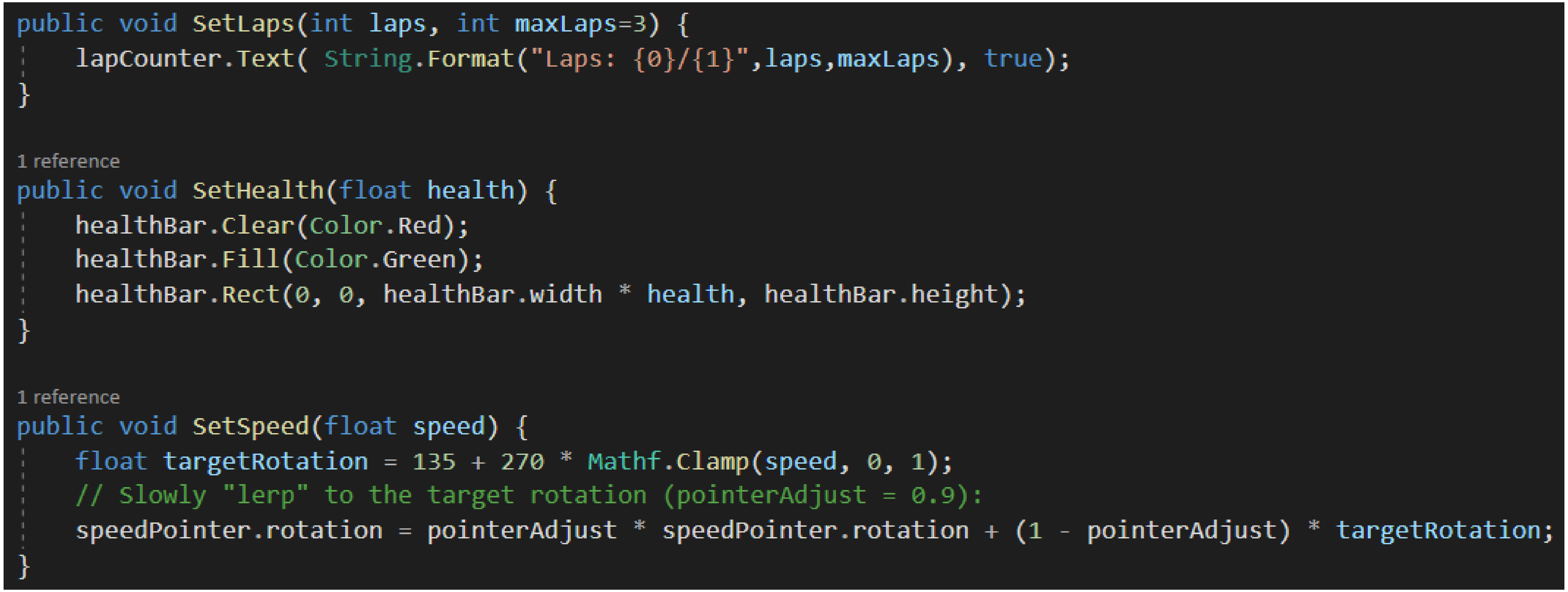
- See the *cheat sheet* on Blackboard for a quick introduction
- See the *class itself*, or the *full documentation* for details
- Now: an example



```
bangers = Utils.LoadFont("bangers.ttf",40);
lapCounter = new EasyDraw(250, 60);
lapCounter.TextFont(bangers);
lapCounter.TextAlign(CenterMode.Min, CenterMode.Center);
lapCounter.Fill(Color.Yellow); // Alternatively: Fill(255,255,0)
lapCounter.Text("Laps: 0/3");
lapCounter.SetXY(20, 20);
AddChild(lapCounter);

healthBar = new EasyDraw(250, 60);
healthBar.ShapeAlign(CenterMode.Min, CenterMode.Min);
healthBar.NoStroke();
healthBar.Fill(Color.DarkGreen);
healthBar.Rect(0, 0, 250, 60);
healthBar.SetXY(game.width - 270, 20);
AddChild(healthBar);

speedoMeter = new EasyDraw(150, 150);
speedoMeter.ShapeAlign(CenterMode.Center, CenterMode.Center);
speedoMeter.StrokeWeight(10);
speedoMeter.Stroke(Color.Red);
speedoMeter.Fill(Color.White);
speedoMeter.Ellipse(75, 75, 140, 140);
speedoMeter.SetOrigin(75, 75);
speedoMeter.SetXY(game.width / 2, game.height - 75);
AddChild(speedoMeter);

speedPointer = new Sprite("laserPurple.png"); // The closest thing to a needle sprite...
speedPointer.SetOrigin(20, speedPointer.height / 2);
speedPointer.scale = 2;
speedoMeter.AddChild(speedPointer);
```



Updating HUD Info



```
public void SetLaps(int laps, int maxLaps=3) {
    lapCounter.Text( String.Format("Laps: {0}/{1}",laps,maxLaps), true);
}

1 reference
public void SetHealth(float health) {
    healthBar.Clear(Color.Red);
    healthBar.Fill(Color.Green);
    healthBar.Rect(0, 0, healthBar.width * health, healthBar.height);
}

1 reference
public void SetSpeed(float speed) {
    float targetRotation = 135 + 270 * Mathf.Clamp(speed, 0, 1);
    // Slowly "lerp" to the target rotation (pointerAdjust = 0.9):
    speedPointer.rotation = pointerAdjust * speedPointer.rotation + (1 - pointerAdjust) * targetRotation;
}
```

These methods are called from Level or from Player

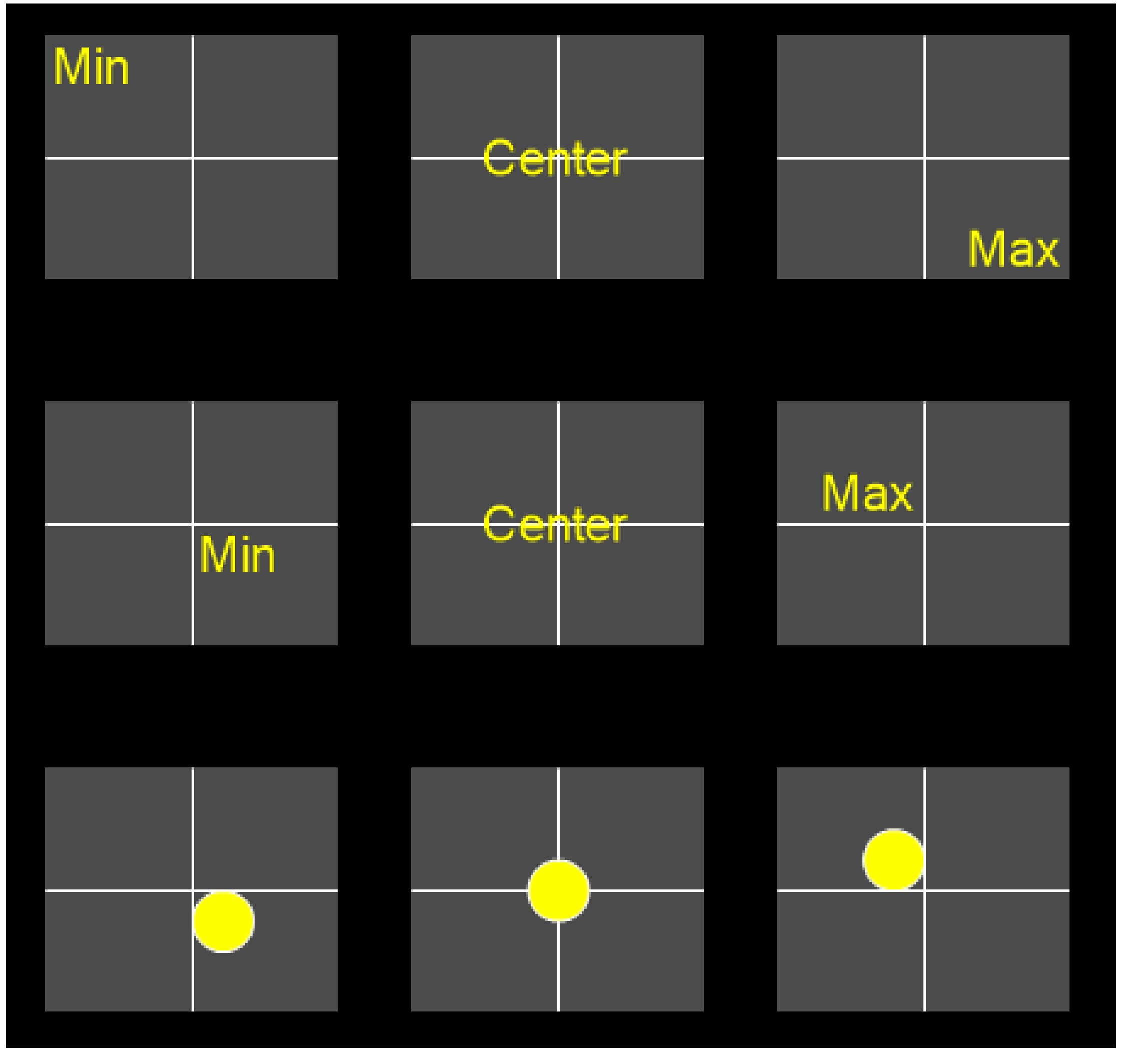
Color and Alignment

- Most EasyDraw methods are self-explanatory

- Note: color values are *RGB(A)*, between 0 and 255. (Red, Green, Blue, Alpha)

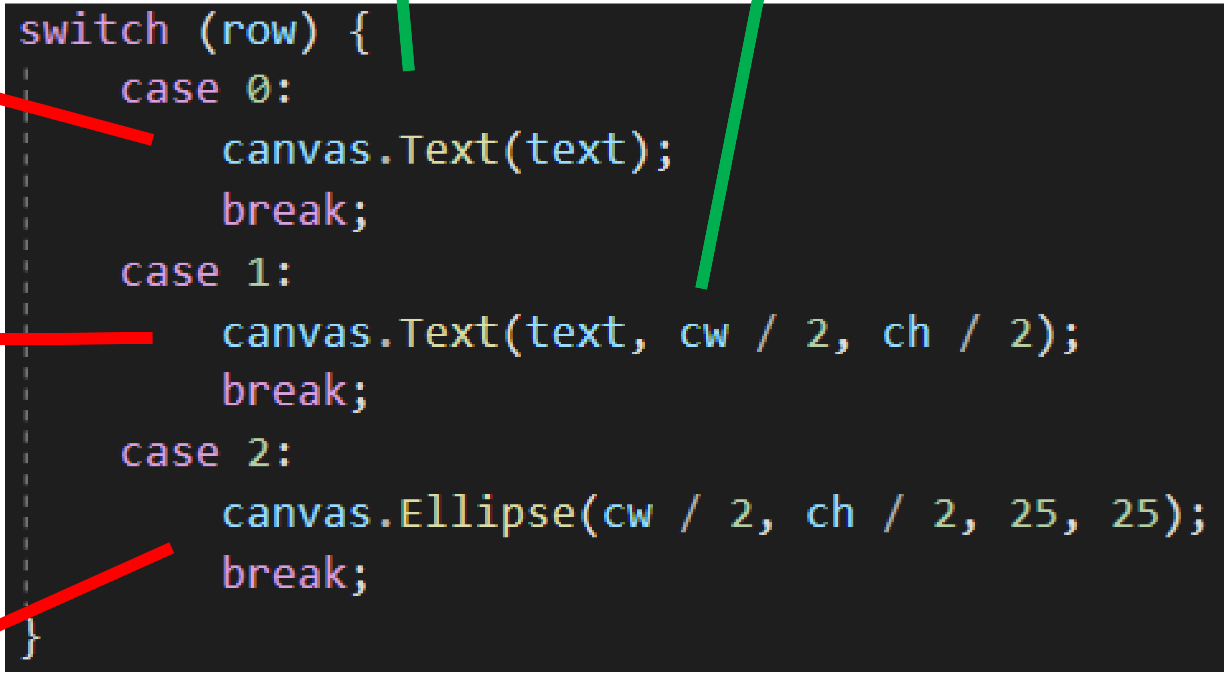
- *TextAlign* and *ShapeAlign* might be a bit confusing: see the code example on Blackboard (next slide)

EasyDraw Alignment



Use case: your EasyDraw is a text element with single string

Use case (example): draw a column of text elements on a single EasyDraw



```
switch (row) {
    case 0:
        canvas.Text(text);
        break;
    case 1:
        canvas.Text(text, cw / 2, ch / 2);
        break;
    case 2:
        canvas.Ellipse(cw / 2, ch / 2, 25, 25);
        break;
}
```

See the EasyDrawAlignment code sample on Blackboard

More on Inheritance

Virtual, override, protected – how, why and when

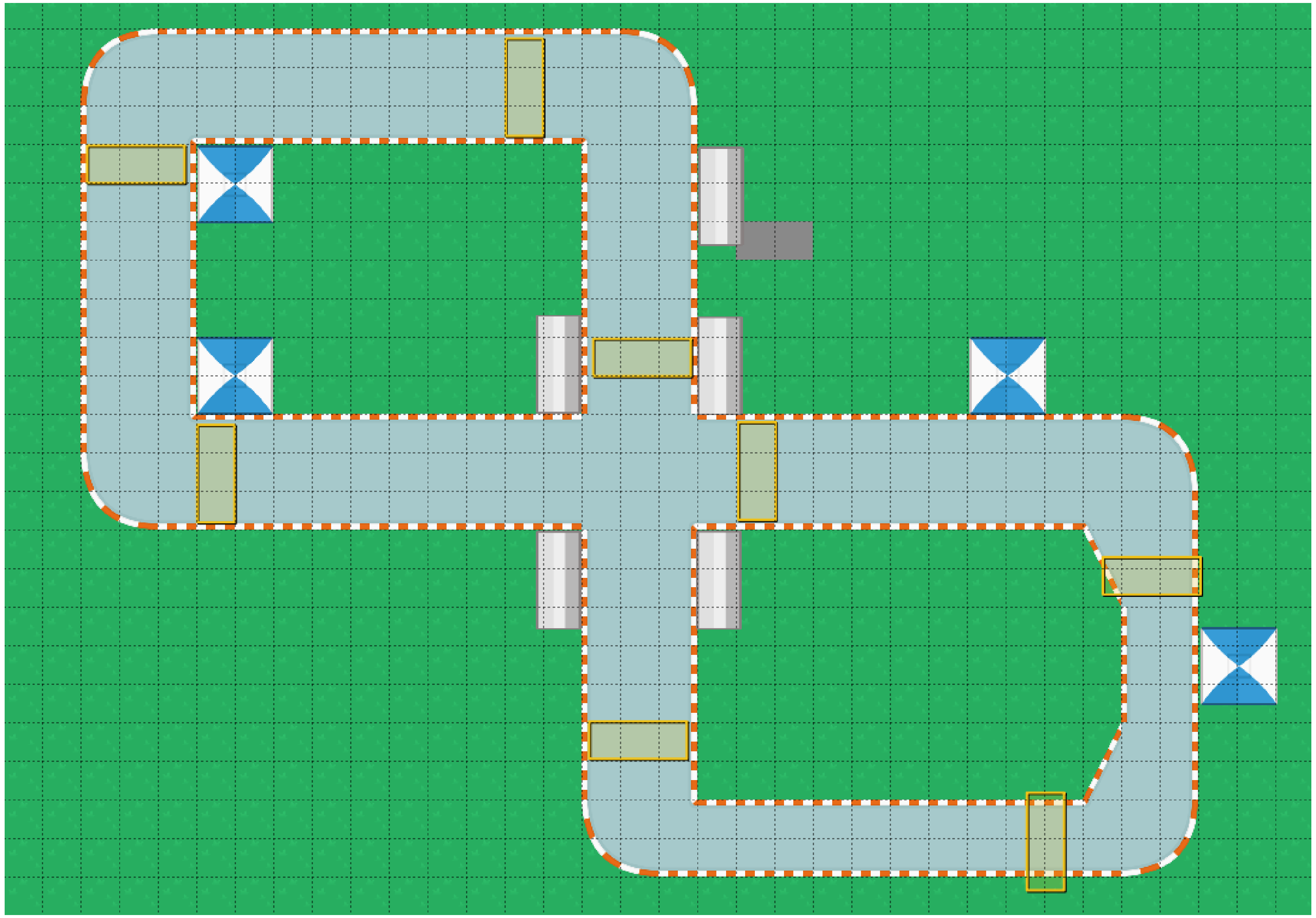
Inheritance

- Today’s example game contains two car types:
  - Controlled by Player
  - Controlled by AI
- If you would put all of this functionality in one class, it becomes large and unwieldy: it’s good to split this up
- This is where *inheritance* is useful
- You’ve seen and used it before (GameObject, Sprite, …), but let’s dig deeper

Inheritance – Class Responsibilities

- *Car* (super class):
  - Contains all the driving physics, collision handling, lap counter *(checkpoints)*
- *CarPlayer*:
  - Uses *keyboard input* to control steering and acceleration
  - Updates the HUD
- *CarAI*:
  - Contains some *basic AI* to control steering and acceleration
  - Basic idea: *waypoints*.
    - Get *angle* to next waypoint for steering.
    - Accelerate when the angle is small, or current speed is low
    - Back up for a few milliseconds after a collision

Tiled: Eight *Waypoints / Checkpoints*



Inheritance: How?

- The *Car* class contains a *Step* method, which calls *GetAcceleration*
- *GetAcceleration* is implemented differently in *CarPlayer* and *CarAI*

- How can we do this?

- Note: a superclass class should *never* contain explicit casts to subclasses! (This leads to unmaintainable code!)

- Solution: *virtual / override*

Example: Refactoring Code Using Inheritance

- *Live demo*
- *See also the code handouts on Blackboard* (InheritanceExample / NoInheritanceExample)

Virtual / Override / Protected

- With the *virtual* keyword, you can define methods that can be overridden in subclasses
- Subclasses: use the *override* keyword
- Method *name*, *return type* and *parameter list* needs to match!
- With the *base* keyword, you can call the original implementation in the subclass (similar to constructors)

- With the *protected* keyword (access modifier), you can make variables and methods accessible by *subclasses* – it’s in between *private* (this class only) and *public* (all classes).

Inheritance: Why?

- *Short, maintainable classes: player inputs* and *AI* code have little in common, and should be in separate classes.
- *Code reuse / Don't Repeat Yourself (DRY):*
  - You don't want to have the same *car physics code* in two classes!
- *Interfaces*:
  - The Game (or Level) only sees *Cars*, and doesn't care about the details of their implementation (player controlled or not)

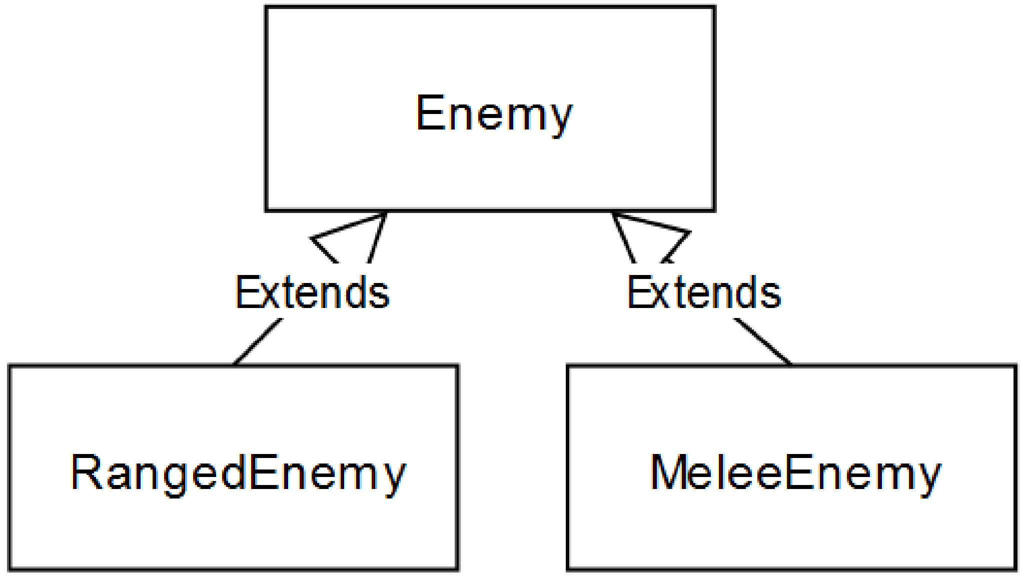
Inheritance: When?

- If you have a two player game (local multiplayer), should you create two classes? (PlayerWASD, PlayerArrows?)
- NO! In this case, the only difference is in the inputs (=*data*), but all the *code* is the same: just set the player inputs!
    - Example: `if (playerIndex==0) leftKey=Key.LEFT; else leftKey = Key.A;`

Inheritance: When?

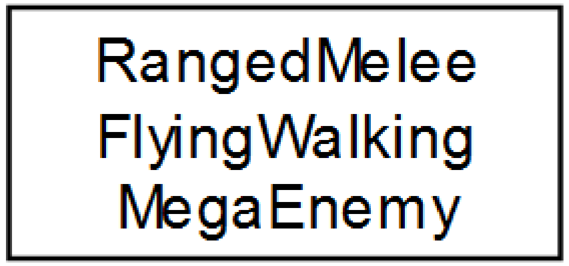
- Sometimes inheritance is the right solution, for instance: you have an *Enemy* class, and enemy types like:
  - *RangedEnemy*, *MeleeEnemy*, *or*
  - *WalkingEnemy*, *FlyingEnemy*

- …but what if you have a *RangedWalkingEnemy, RangedFlyingEnemy, MeleeWalkingEnemy,* and *MeleeFlyingEnemy?*



This is good code architecture

Not Good!

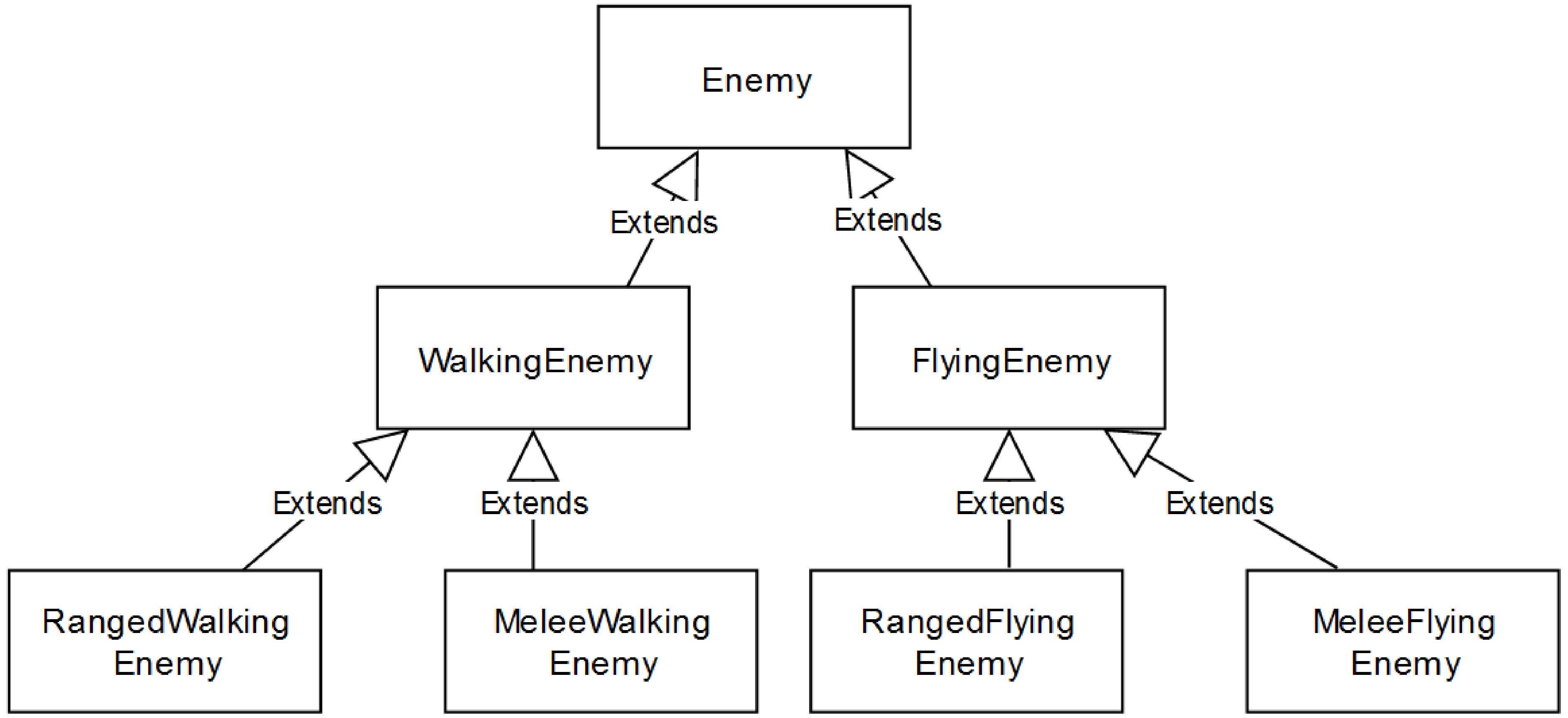


1000 lines of code in one class?

Bad cohesion

Not: *Single Responsibility*

Not Great Either



Note: the Ranged code and the Melee code are still at two places!

Worse

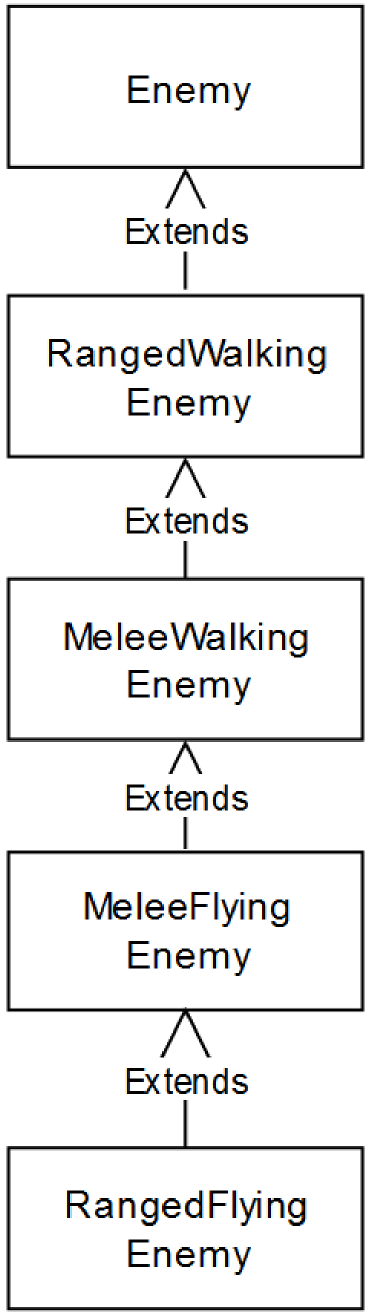
- There's very little code duplication here, which is good…
- …but still such code is not maintainable (what if you want to add a third attack behavior??)
- *Guideline: only use inheritance for is-a relationships*
  - A RangedEnemy *is an* Enemy *is a* Sprite, etc. (=good)
  - But a RangedFlyingEnemy *is not* a MeleeFlyingEnemy (=bad)

Introduce *ranged* (*attack*) and *walking* (*movement*) behavior

Override *attack* behavior: *melee*

Override *movement* behavior: *flying*

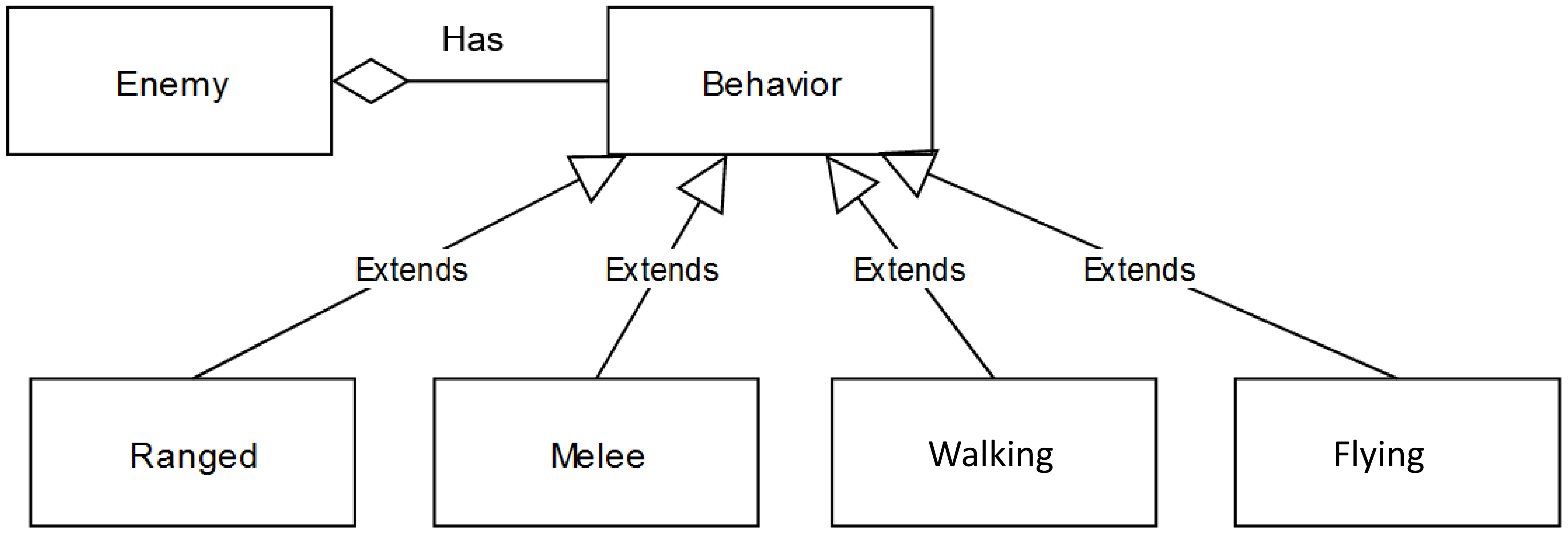
Override *attack* behavior again: back to *ranged*



Solution

- So what’s the answer here…?
- In this case, you can create a *Behavior* super class. Subclasses:
  - Ranged
  - Melee
  - Flying
  - Walking
- An Enemy can *have* any number of behaviors
- This is a component based architecture (like used by Unity) → Note: still built on OOP techniques and inheritance!

Behaviors / Components

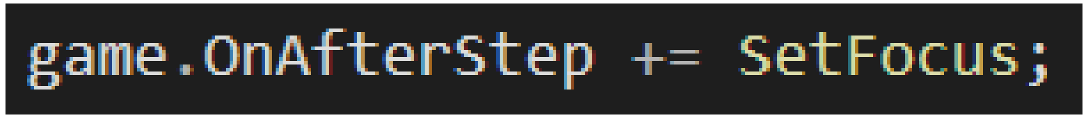


Final Example

It’s all coming together now

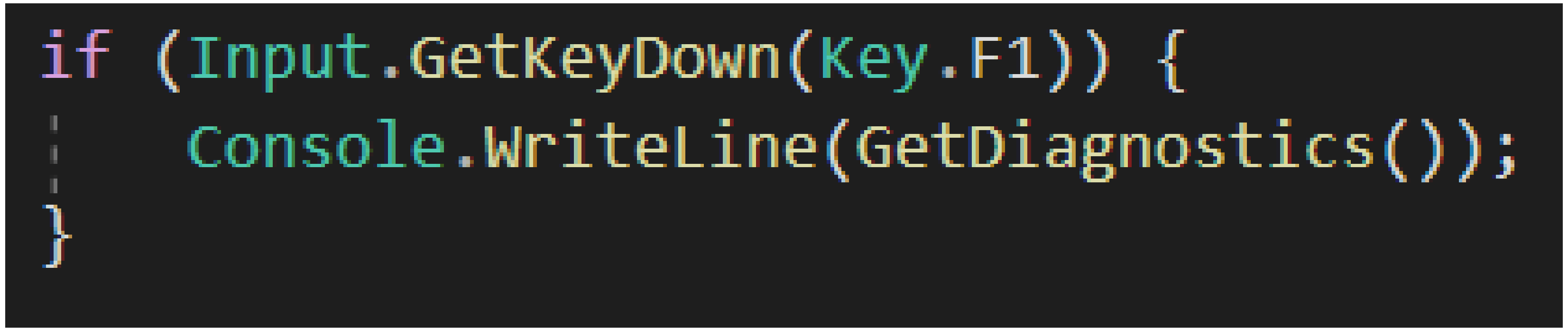
Perfect Camera Focus

- Suppose we do the level scrolling (*SetFocus* method) using *OnAfterStep*, to guarantee the correct position (after the player has moved!)
- Level:



```
game.OnAfterStep += SetFocus;
```

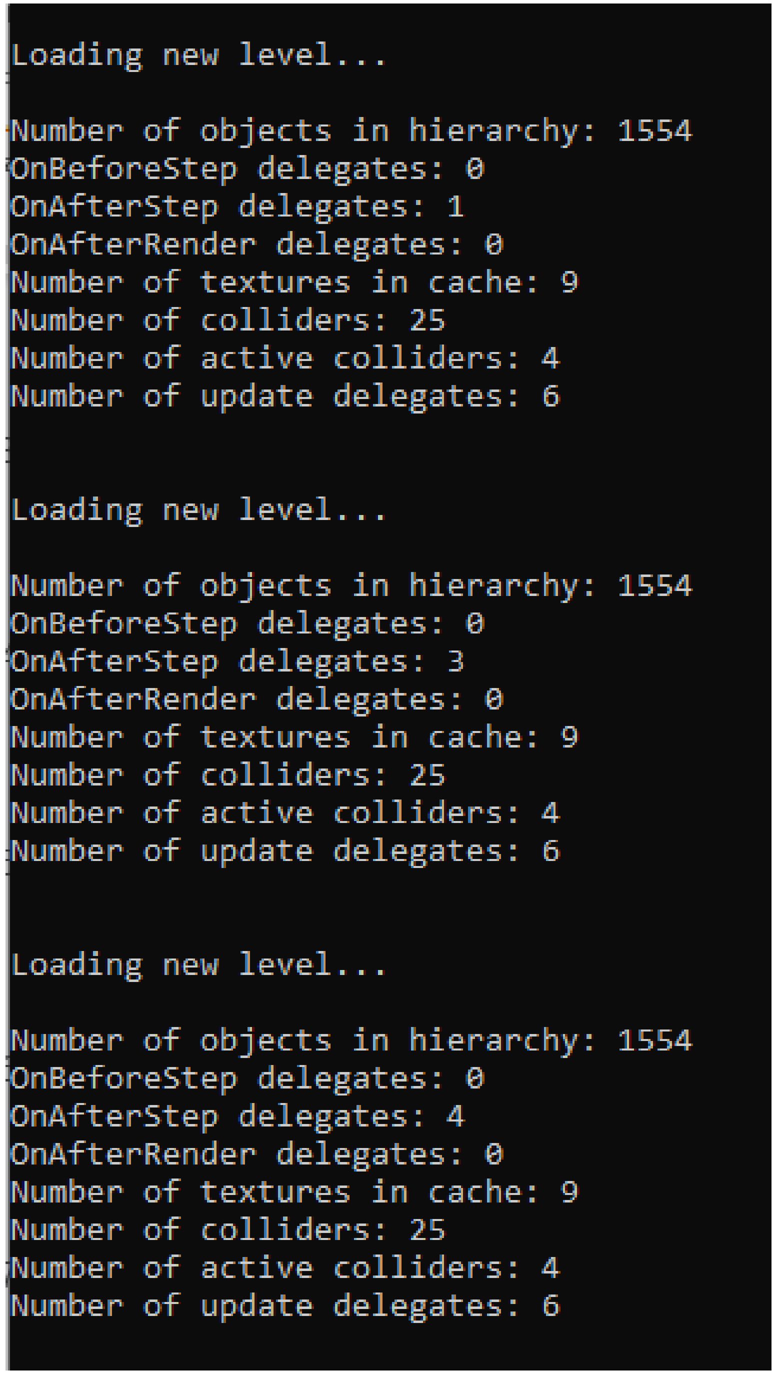
- Some performance checking code in MyGame:



```
if (Input.GetKeyDown(Key.F1)) {
    Console.WriteLine(GetDiagnostics());
}
```

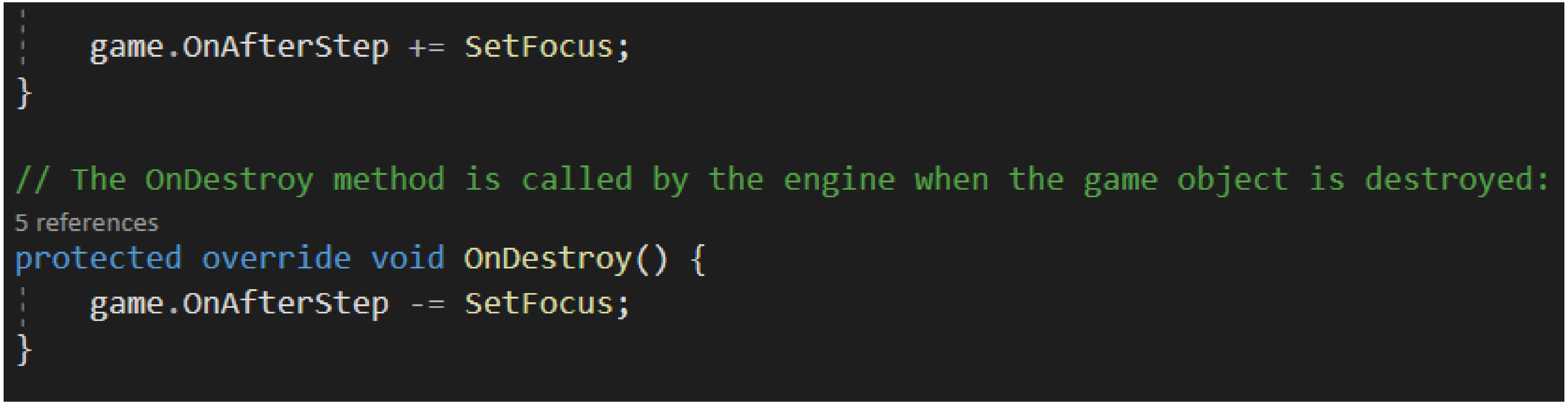
Problem

- What’s going wrong here?
- We forgot to *unsubscribe* to the OnAfterStep event!
- (The lifetime of the Game is longer than that of the Level objects!)
- The best place to unsubscribe is in the *OnDestroy* method of Level



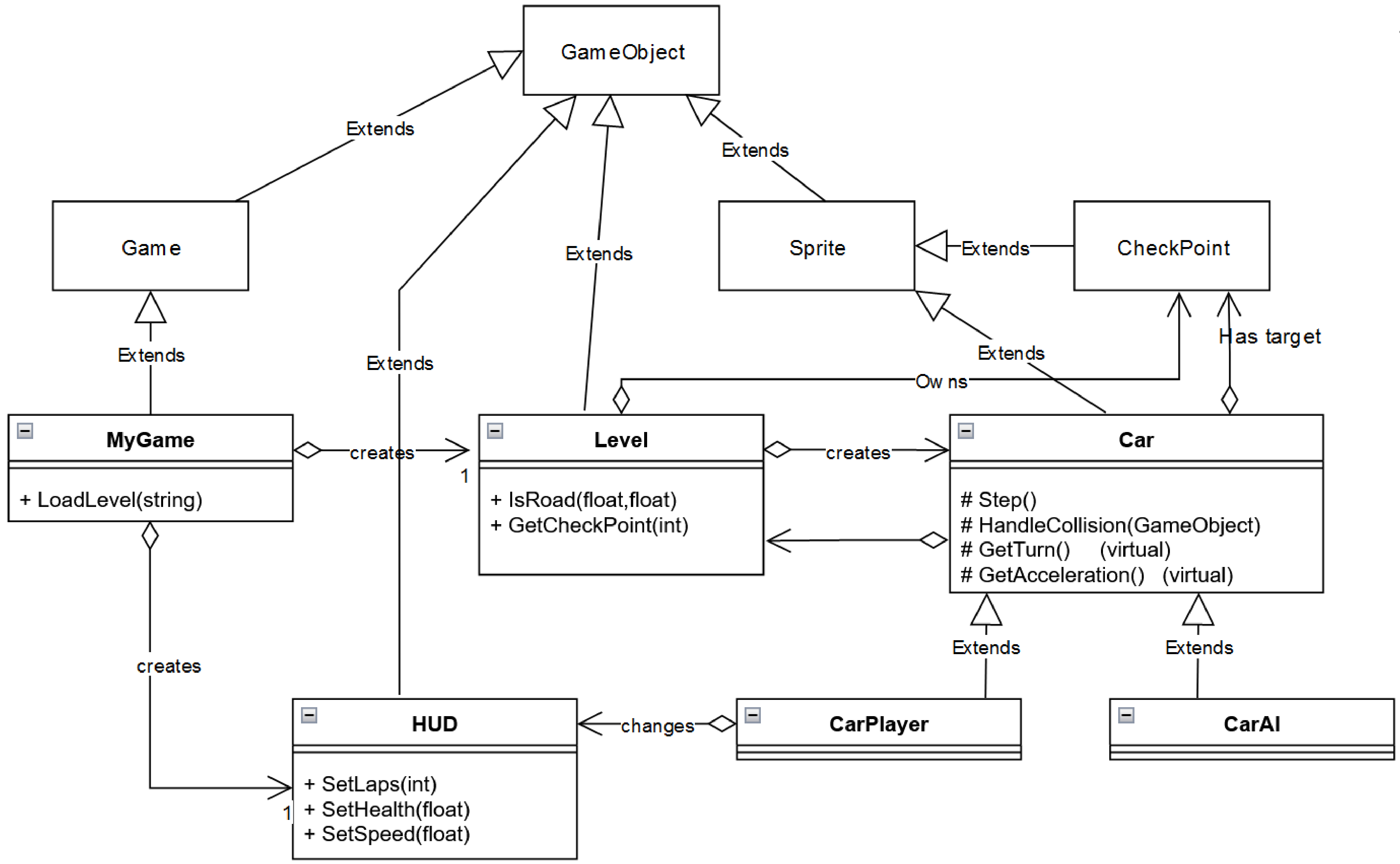
```
Loading new level...

Number of objects in hierarchy: 1554
OnBeforeStep delegates: 0
OnAfterStep delegates: 1
OnAfterRender delegates: 0
Number of textures in cache: 9
Number of colliders: 25
Number of active colliders: 4
Number of update delegates: 6


Loading new level...

Number of objects in hierarchy: 1554
OnBeforeStep delegates: 0
OnAfterStep delegates: 3
OnAfterRender delegates: 0
Number of textures in cache: 9
Number of colliders: 25
Number of active colliders: 4
Number of update delegates: 6


Loading new level...

Number of objects in hierarchy: 1554
OnBeforeStep delegates: 0
OnAfterStep delegates: 4
OnAfterRender delegates: 0
Number of textures in cache: 9
Number of colliders: 25
Number of active colliders: 4
Number of update delegates: 6
```

OnDestroy



```
    game.OnAfterStep += SetFocus;
}

// The OnDestroy method is called by the engine when the game object is destroyed:
5 references
protected override void OnDestroy() {
    game.OnAfterStep -= SetFocus;
}
```

Conclusion / Summary



This *class diagram* was made with *draw.io*

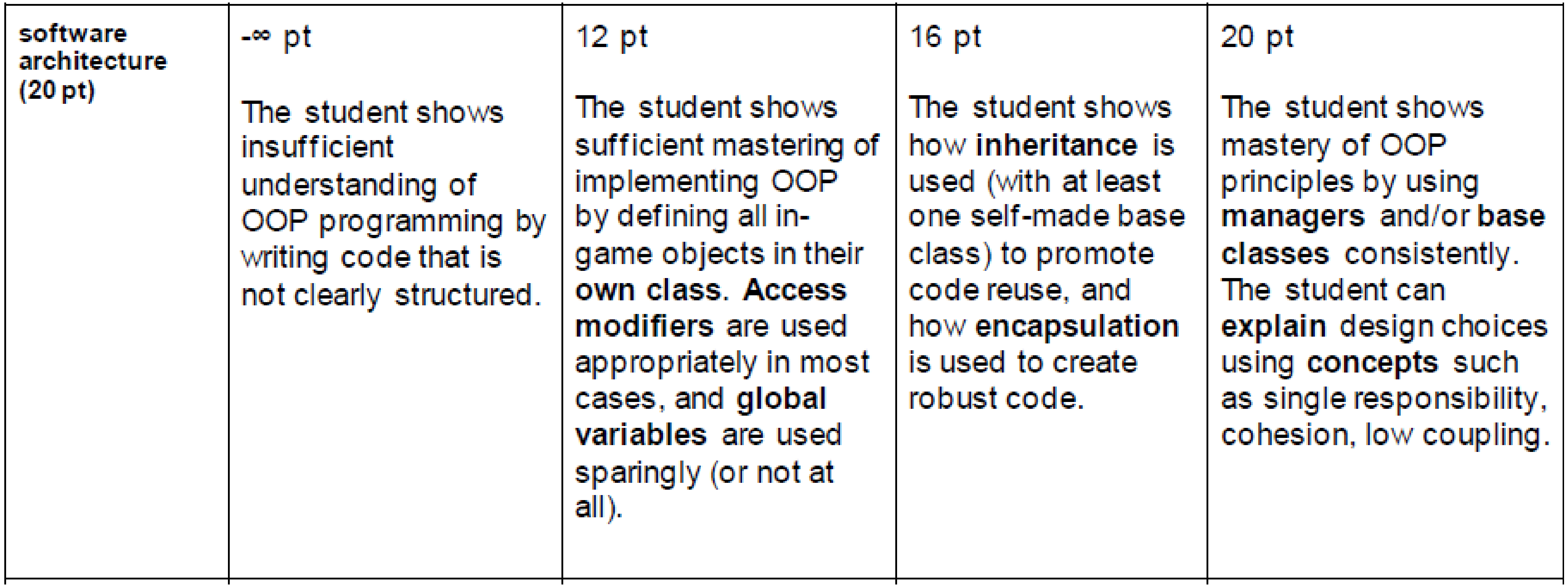
Notation:
+ public
# protected
(private methods
are omitted in th
diagram!)

Summary

Today we learned about:

- How to cause and fix bugs related to level loading
- Events (and delegates): game.OnAfterStep
- Creating a user interface: EasyDraw, HUD class
- Inheritance (an OOP pillar):
  - *How:* virtual, override, base, protected
  - *When/Why:* some *Object Oriented Design* guidelines

A Look at the Grading Criteria



software architecture (20 pt)	-∞ pt The student shows insufficient understanding of OOP programming by writing code that is not clearly structured.	12 pt The student shows sufficient mastering of implementing OOP by defining all in-game objects in their **own class. Access modifiers** are used appropriately in most cases, and **global variables** are used sparingly (or not at all).	16 pt The student shows how **inheritance** is used (with at least one self-made base class) to promote code reuse, and how **encapsulation** is used to create robust code.	20 pt The student shows mastery of OOP principles by using **managers** and/or **base classes** consistently. The student can **explain** design choices using **concepts** such as single responsibility, cohesion, low coupling.

This should start to make sense now….

During the Lab

- Add user interface elements to your game (suggested: a HUD class, containing EasyDraws and/or (Animation)Sprites)
- Make sure your level loading does not contain bugs!
- Get feedback on your code

Optional:

- Think about how you can use inheritance to improve your code

Next Week

Making it *juicy:* adding visual effects, sound effects