**Programming Design Patterns for Unity Development**

1. **Object Pooling:**

Objectcycling: Instantiate and destroy objects very frequently

**Problems:**

1. Instantiating new objects costs performance and fragments the RAM
2. Destroying objects also costs performance because of the garbage collector (THIS IS A MAJOR PROBLEM)

Games where we most use objectcycling:

* Bullet hell
* RTS
* Shooter
* Endless Runner

**Solution:**

Instantiate Objects only once and instead of destroying them, just disable them and reuse them.

**Pros:**

* Better performance at runtime

**Cons:**

* Longer loading time (pool needs to be filled at runtime)
* Takes time to implement

Implementation:

1. Instantiate certain amount of objects
2. Deactivate all these objects
3. Save them in a Collection (Array/Queue)

When an object is needed?

* Instead of Instantiating the object, just actívate one from the pool.

When an object is no longer needed?

* Deactivate it and put it back into the pool

Code Sample:

///In PoolMethod

//Initializing Method

public void InitPool(int poolSize)

{

\_pool=new List<GameObject>();

for(int i=0;i<poolSize;i++)

{

\_pool.Add(Instantiate(\_objectToPoolPrefab));

\_pool[i].SetActive(false);

}

}

//Getting and returning objects from the pool

//Get from pool

public GameObject GetObjFromPool(Vector3 pos, Quaternion rot)

{

GameObject newBullet=\_pool[\_pool.Count-1];

newBullet.SetActive(true);

newBullet.transform.position=targetPos;

newBullet.transform.rotation=rot;

\_pool.RemoveAt(\_pool.Count-1);

return newBullet;

}

//Return to pool

public void ReturnObjToPool(GameObject go)

{

go.SetActive(false);

\_pool.Add(go);

}

///

//Referencing in main Method where you can use

//singleton to call the objectPool

public ObjectPool \_objectPool;

//This is like instante but better in performance runtime

GameObject newBullet=\_objectPool.GetObjFromPool(transform.position,transform.rotation);

//Similar to Destroy but better in performance

\_objectPool.ReturnObjToPool(newBullet);

**Queue Theory**

* Collection
* **"Works like a waiting line"**
* First in, First out

1. Enqueue: Set to the back of the collection
   * Means to put something in the queue
2. Dequeue: Put out something from the front
   * Put out of the queue an object
3. **Flyweight:**

Problems In RAM:

* + Huge amounts of objects flood the RAM
  + This should be optimized!

What happens in the RAM?

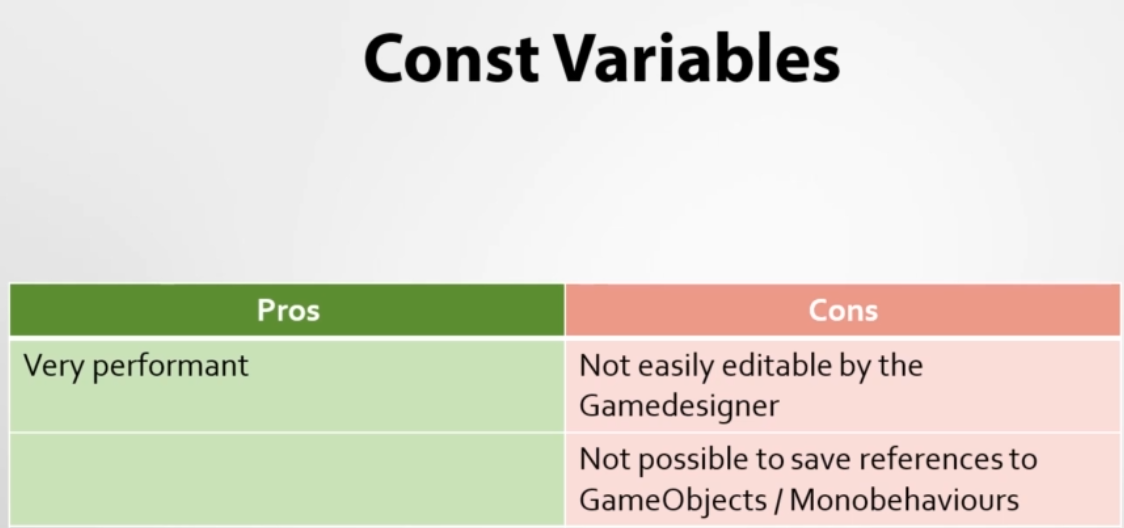
It fills with repeated variables that can be generalized.

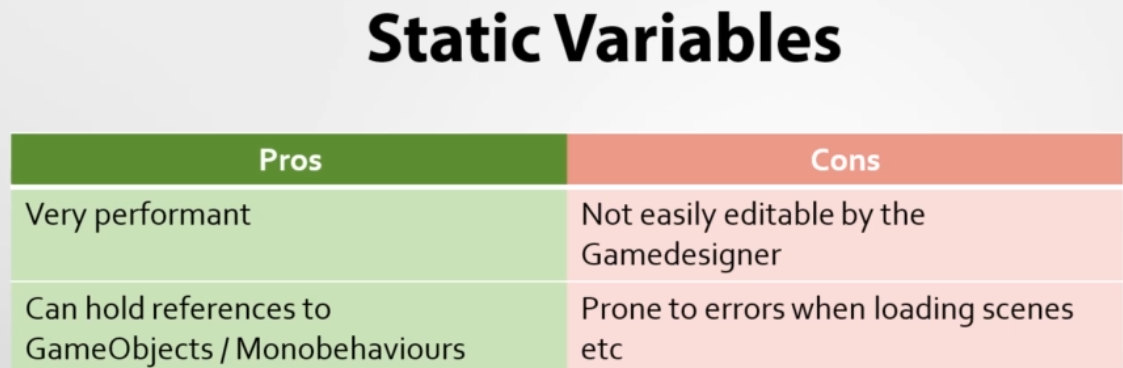
**Example:**

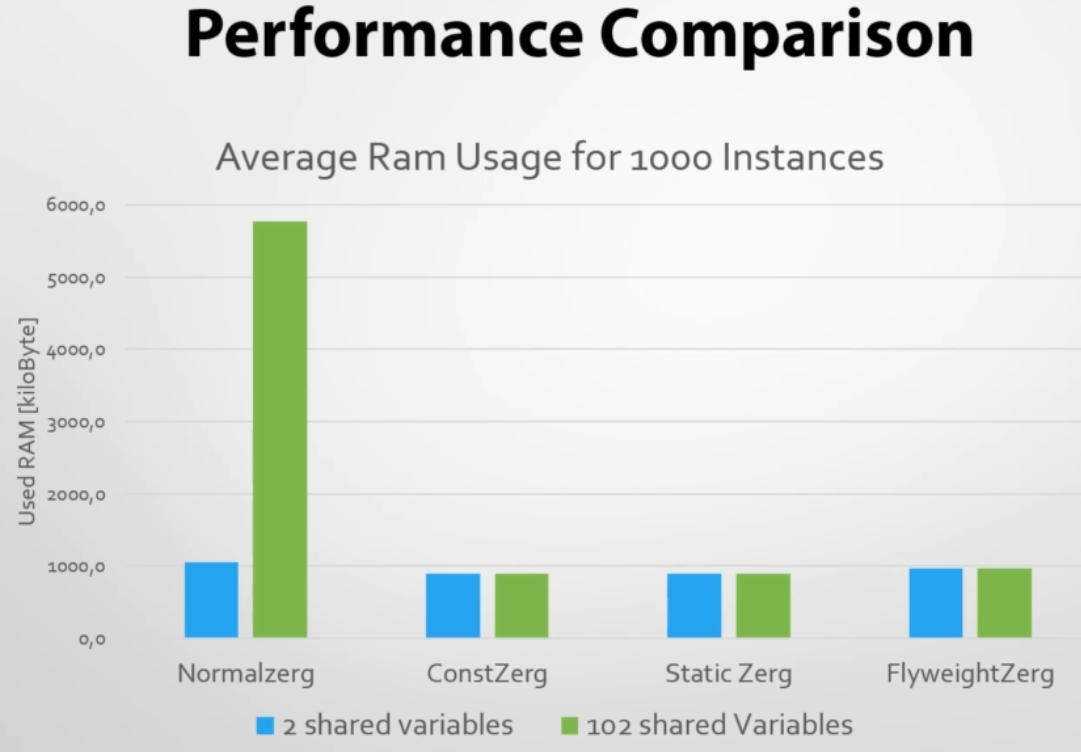
Zerling class:

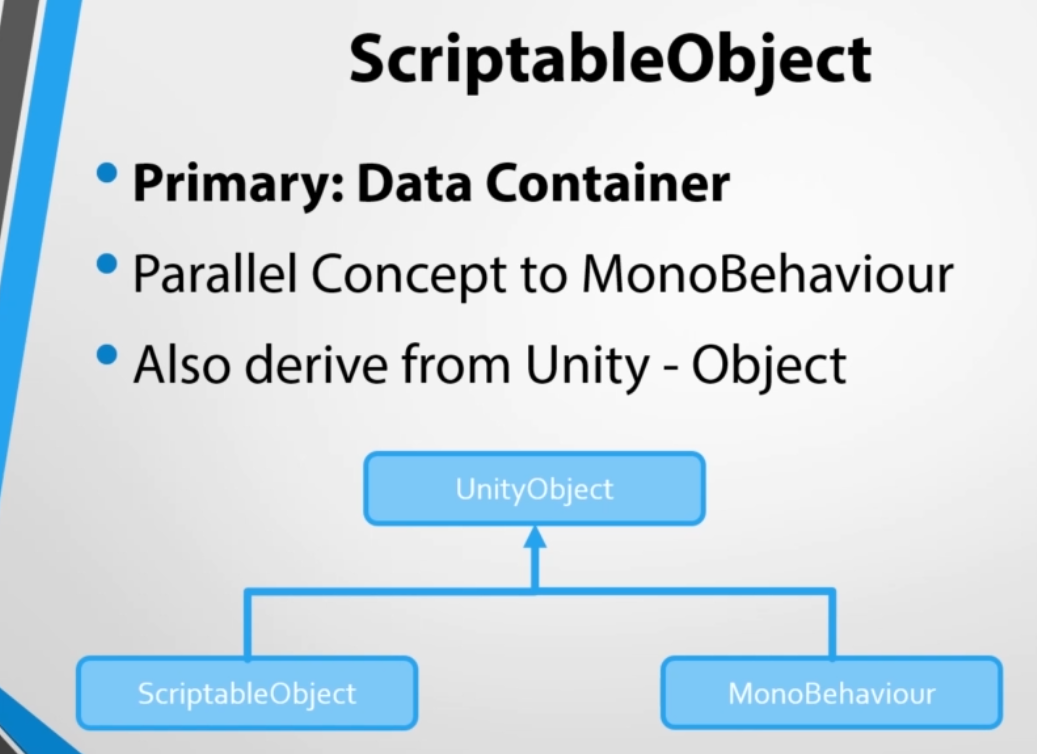
* + int curHp;
  + int maxHp;
  + float movSpeed;

Here we can see that we can generalize some of the variables like, maxHp and movSpeed. Why? Because every Zerling have a maxHp and a movSpeed.







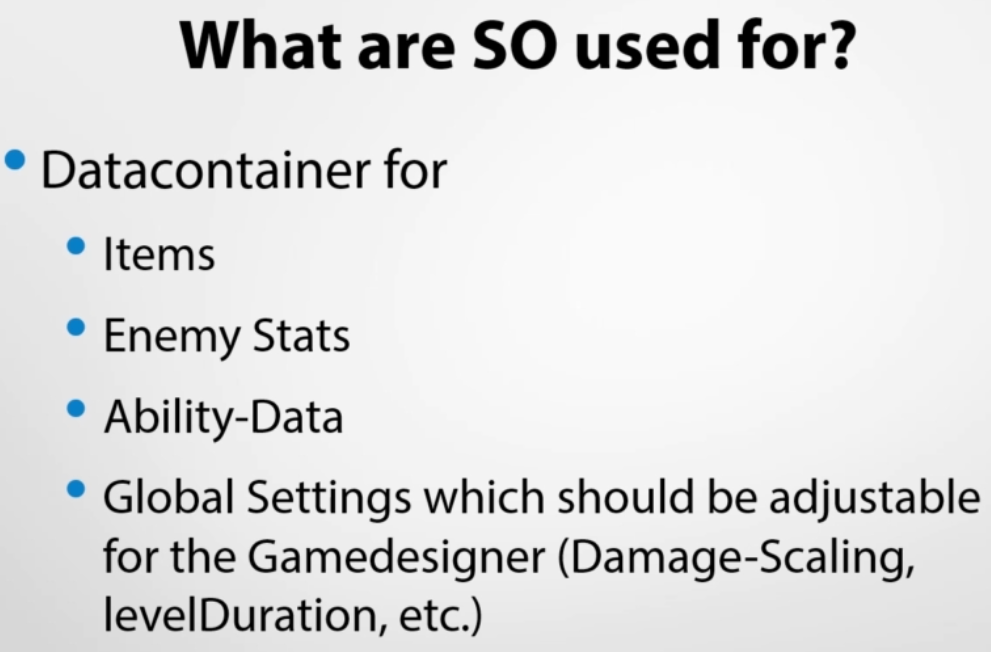


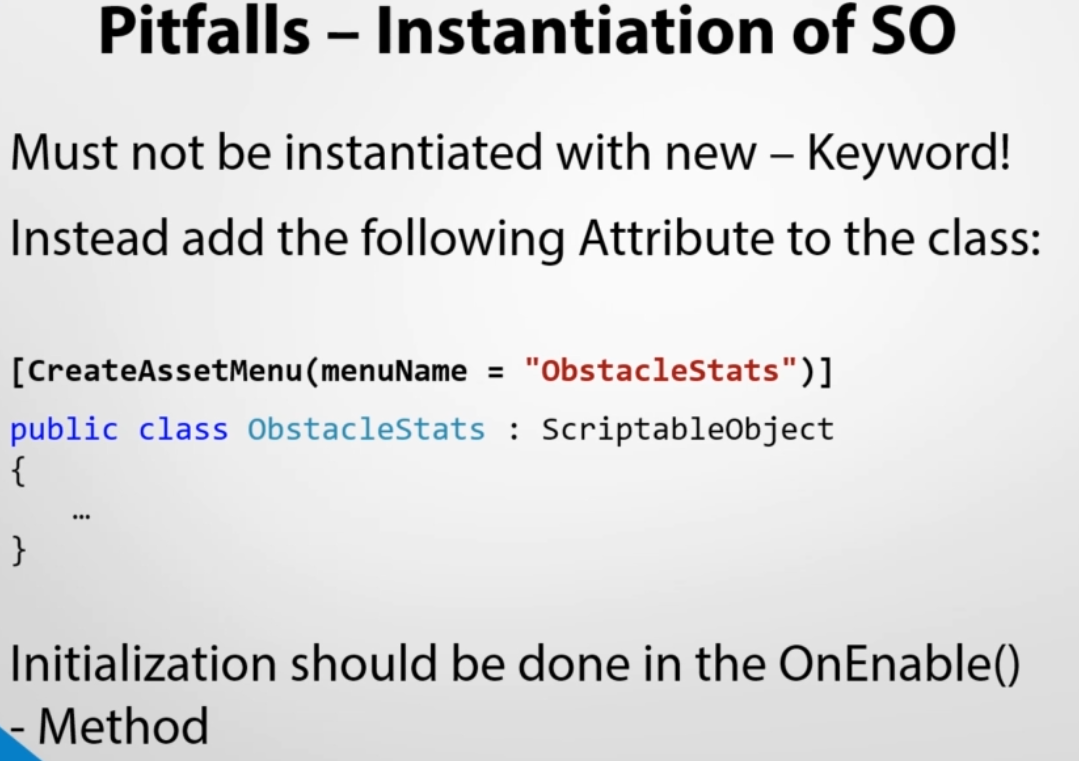
\*Monobehaviour is more about functionality of the game.

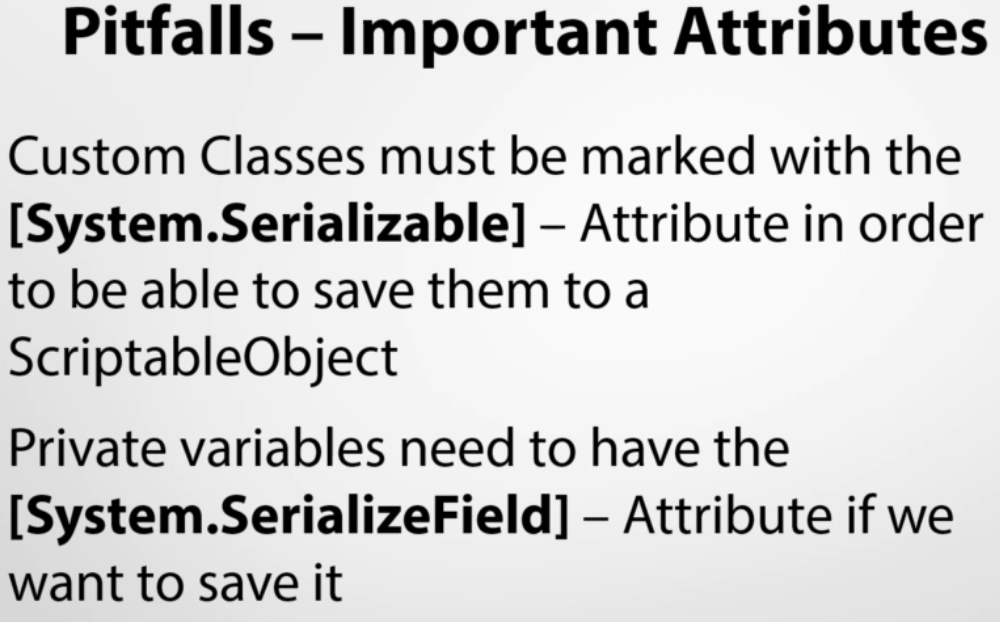
\*ScriptableObject is more the data of the game.

-ScriptableObjects are in a folder, not in the scene.









Sample:

[CreateAssetMenu(menuName=”ItemPrototype”)]

public class ItemPrototype:ScriptableObject

{

public string \_itemName;

public int \_value;

public float \_weight;

public StatBlock \_stats;

}

[System.Serializable]

public class StatBlock

{

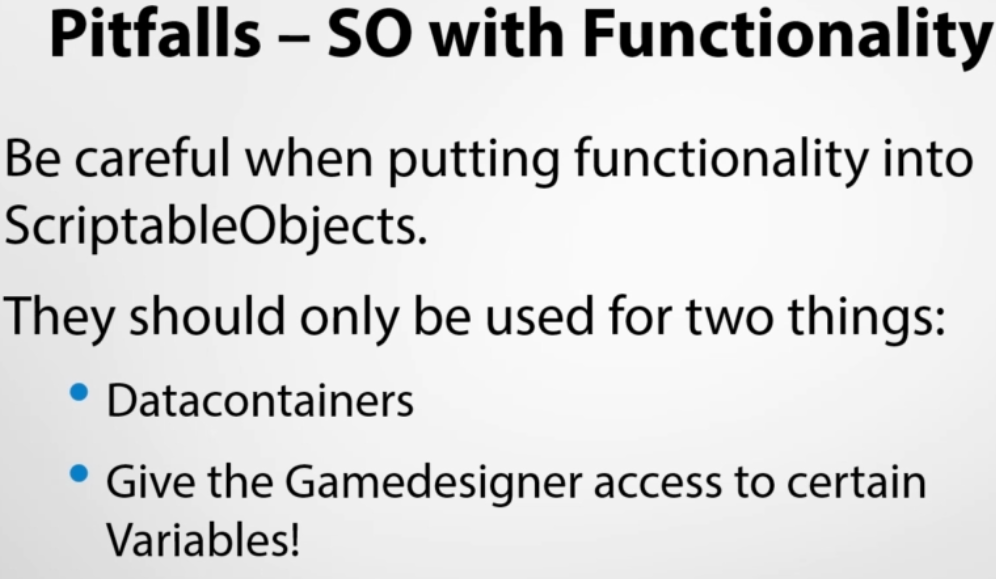
public int strength;

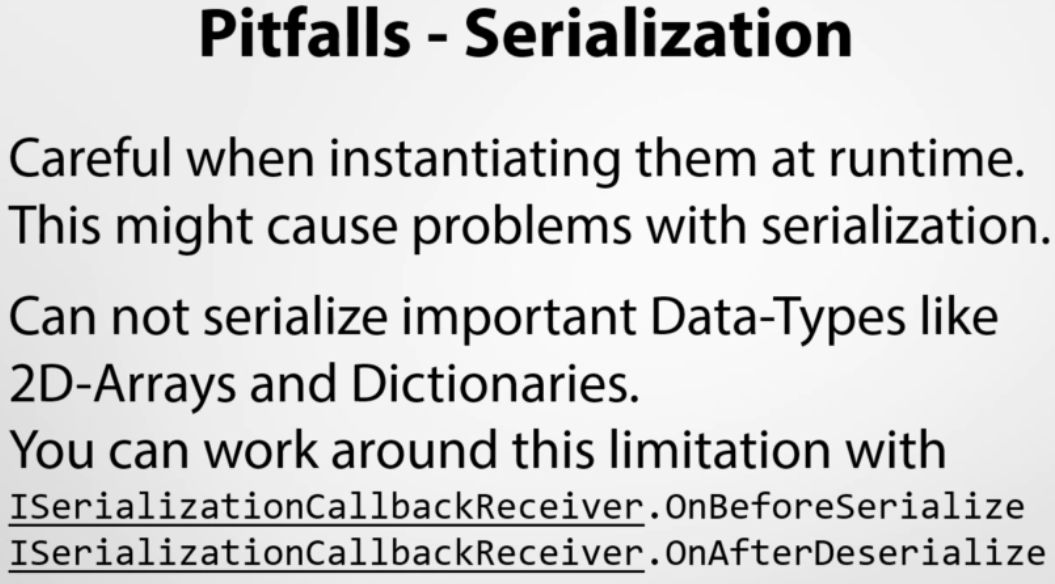
public int dexterity;

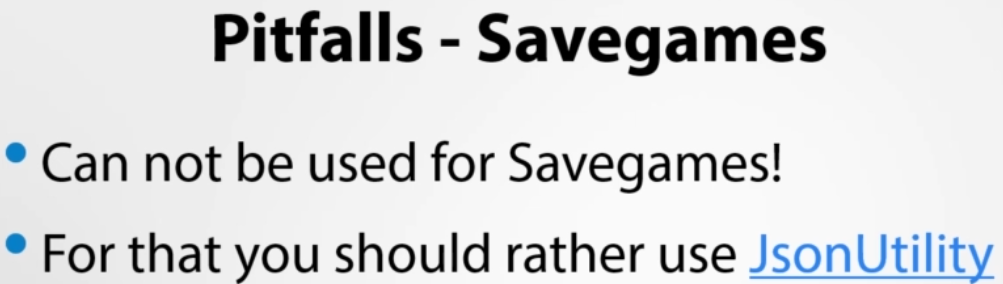
public int intelligence;

}

//Note: Only public variables shows up in the menú  
//Note 2: System.Serializable doesn’t show just class type, can show structs too.









Sample:

[CreateAssetMenu(fileName=”defaultGamePrefs”,menuName=”GamePreferences”)]

public class GamePreferences:ScriptableObject{

private static GamePreferences \_myInstance;

public static GamePreferences \_MyInstance

{

get{

if(\_myInstance==null){

\_myInstance=Resources.Load<GamePreferences>(“GamePreferences”);

}

return \_myInstance;

}

}

public float \_startSpeed;

public float \_speedAcceleration;

public float \_obstacleSpawnCd;

}

//Note: Must use soft Singleton for ScriptableObjects

//Resources folder is where we must have ScriptableObjects that we Will charge

1. **Command**

Used for:

* Controllermapping.
* Queue Actions in turnbased strategy games.
* Creaste History of Actions
* Undo-Functionality

Examples:

i) ControllerMapping

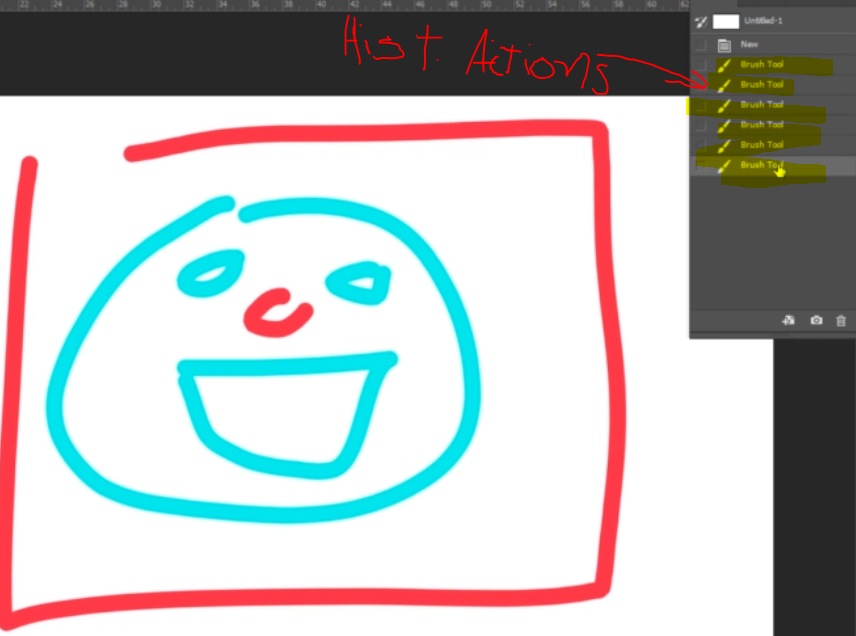


To change controller mapping at runtime.

ii) Realtime strategy games:

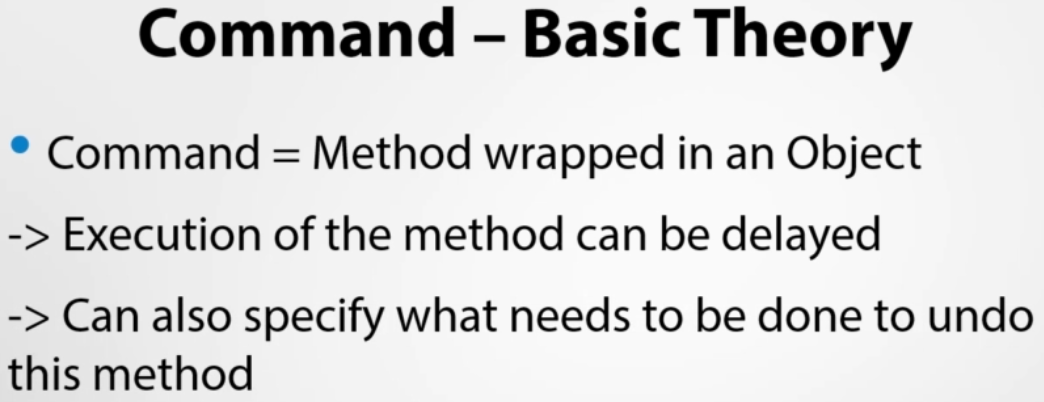
* Queue of actions we want a unit do.

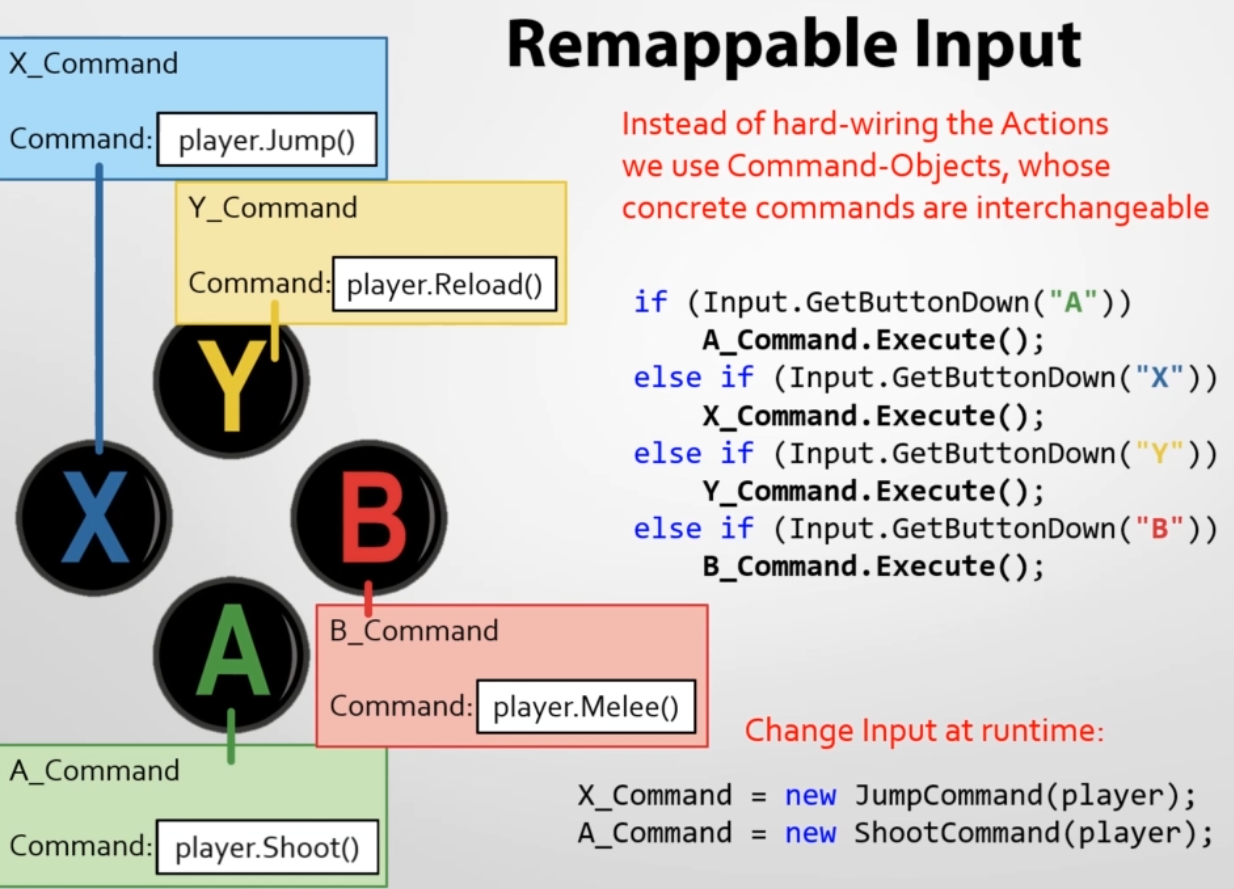
iii) History of actions:

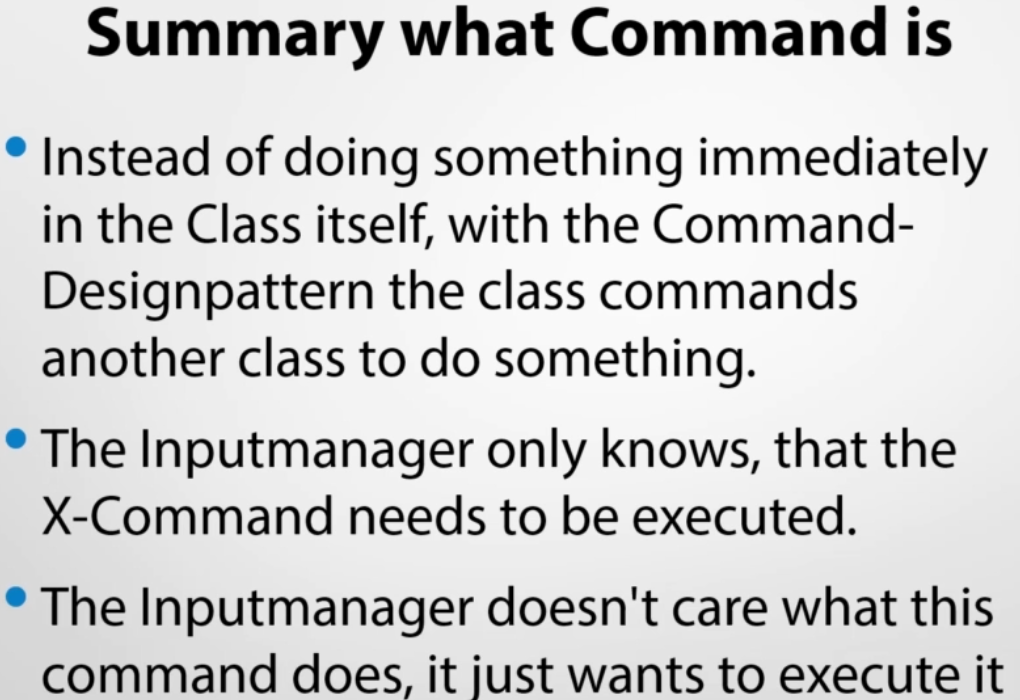


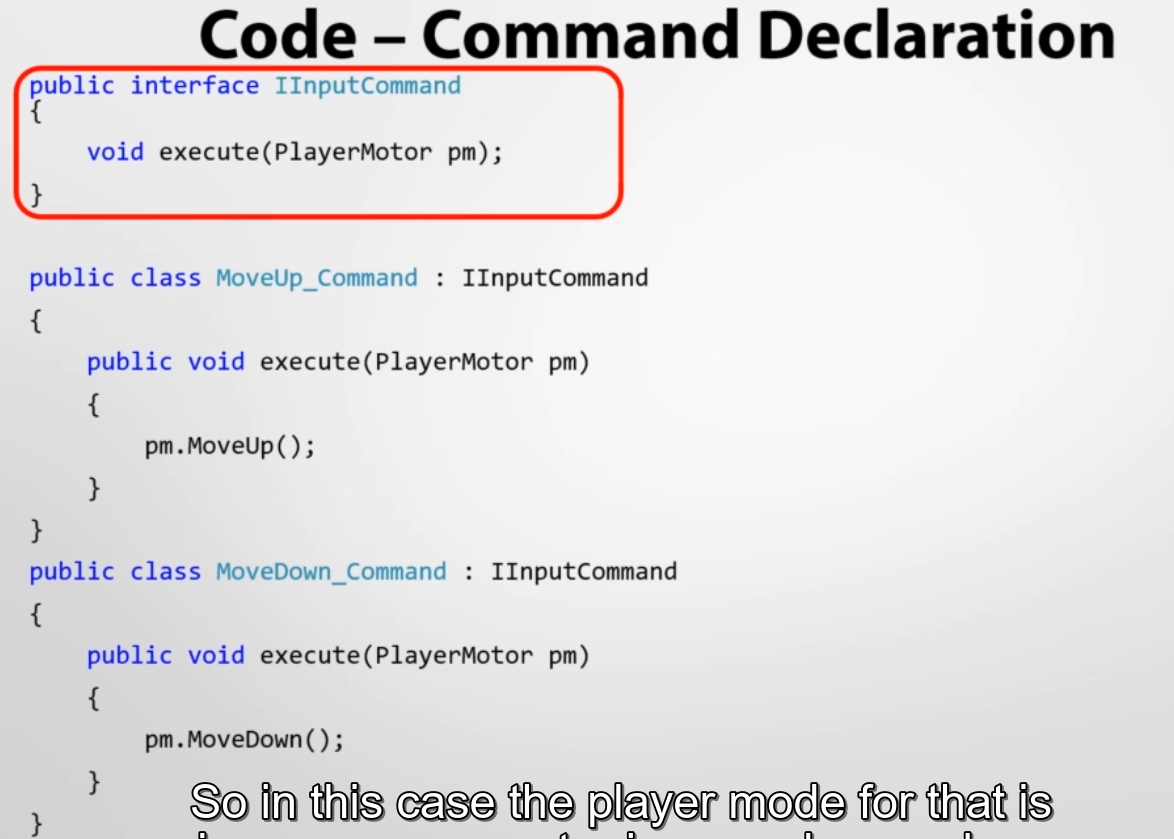
iv) Undo Actions (like ctrl + z or a button to undo movement in a turned base strategy game).

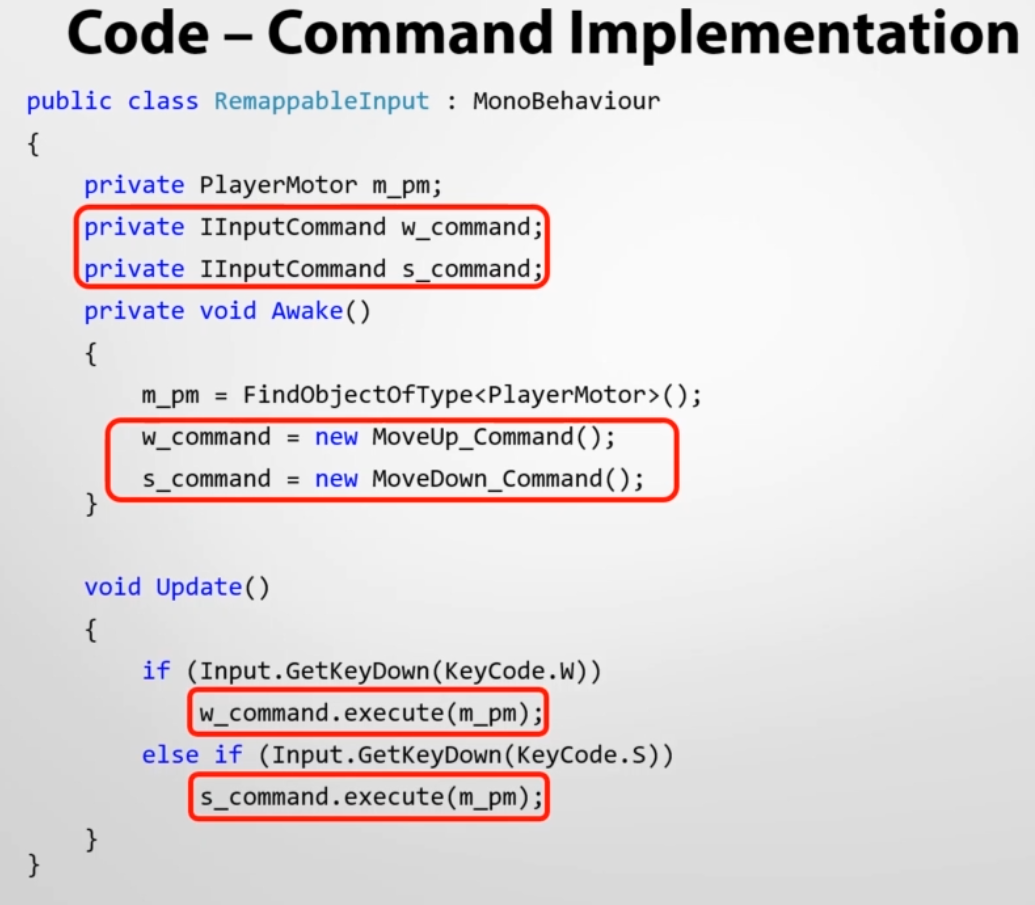
**Remapping Controls**

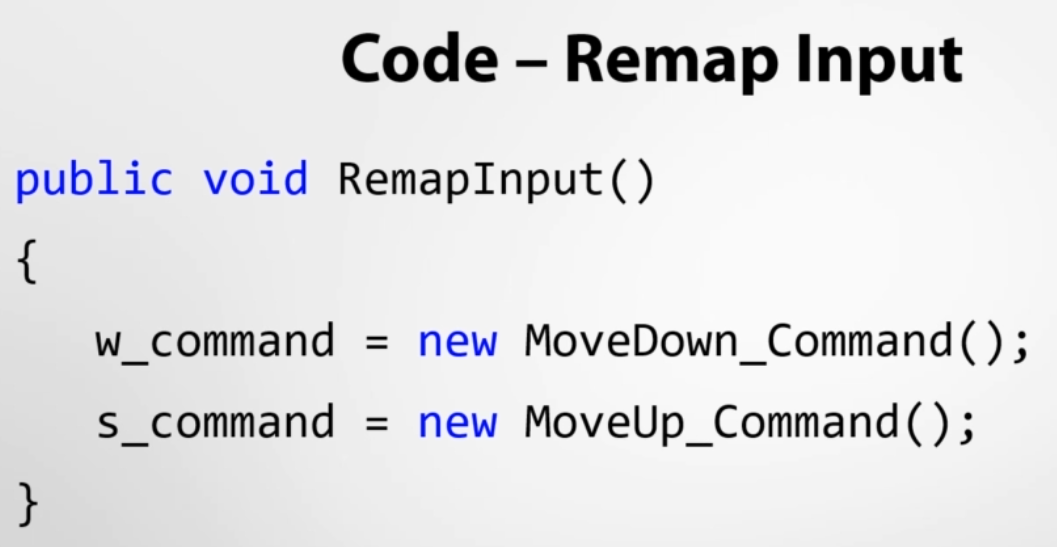




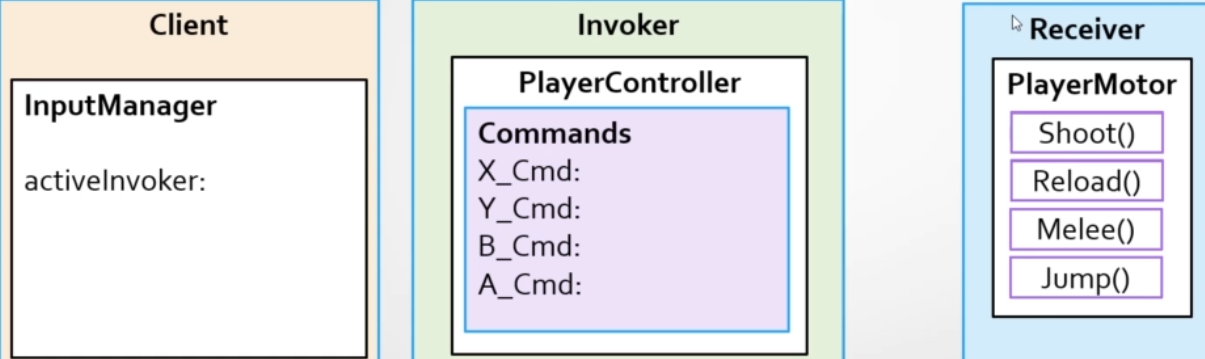




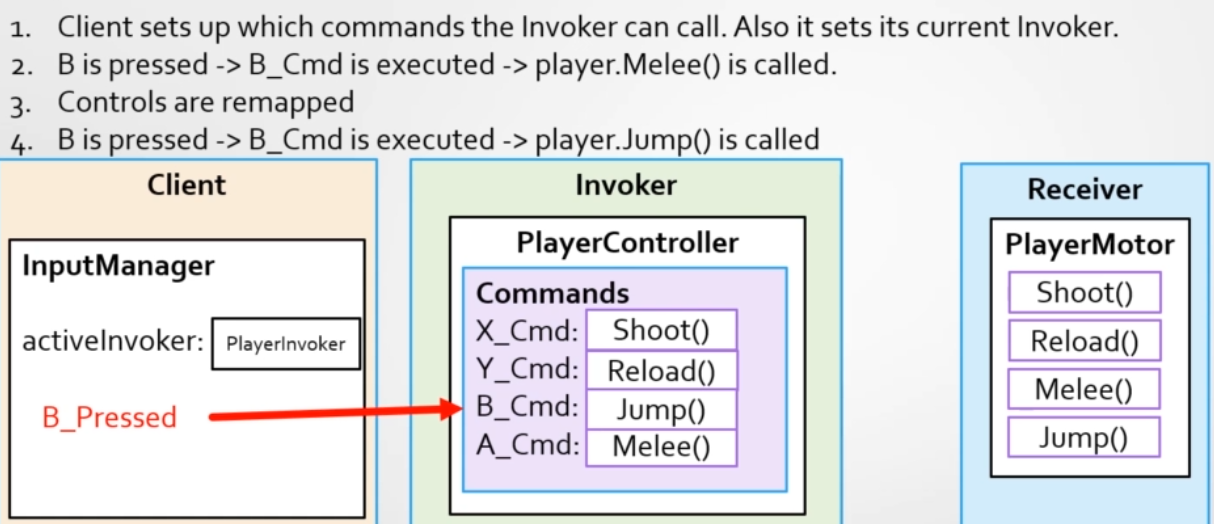




**Formal Definition**







**Queue Commands.**

