Tópicos Especiais em Engenharia de Software (Jogos II)

Aula 04 – Scripting

Edirlei Soares de Lima
<edirlei@iprj.uerj.br>

Unity 3D: Scripting

- A Unity oferece um ambiente completo de programação baseado em scripts (C#, JavaScript).
  - Introdução ao C#
  - Scripts como Components
  - Introdução a Orientação a Objetos
  - MonoBehaviour (Start, Update, ...)
  - Estruturas Condicionais
  - Estruturas de Repetição
  - Vetores, Matrizes, Listas e Dicionários
  - Herança, Polimorfismo, Encapsulamento, Composição

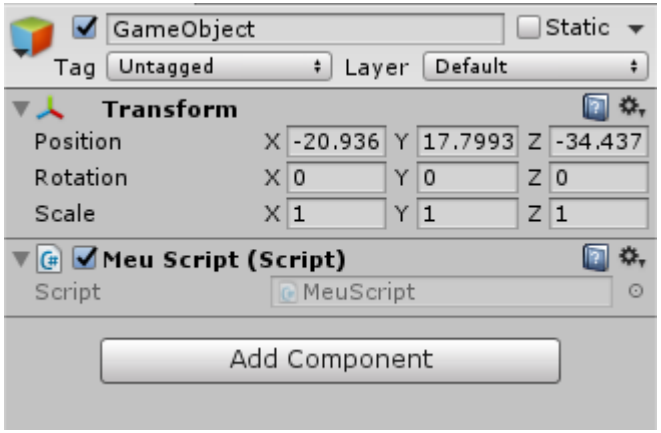
Unity 3D: Scripting

- **Criar um novo script:** Assets -> Create -> C# Script

```
using UnityEngine;
using System.Collections;

public class MeuScript : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }
}
```

Unity 3D: Scripting

- Um script define um **Component**.
  - Nenhuma linha de código de um script é executada sem que ele seja primeiramente associado a um GameObject.



  - Um mesmo script pode ser associado a múltiplos objetos. Cada associação, cria uma nova instancia do script.

“Hello World” na Unity

```
using UnityEngine;
using System.Collections;

public class MeuScript : MonoBehaviour{

    void Start ()
    {
        Debug.Log("Hello World!");
    }

}
```

O comando `Debug.Log` é usado para escrever um texto no console de saída da Unity. É uma boa forma de depurar manualmente a execução dos scripts.

Funções de Evento

- Programar na Unity envolve a implementação de diversas **funções de evento** (*callbacks*). Essas funções são executadas automaticamente em determinados momentos de acordo com as suas funcionalidades.

- Exemplo:

```
void Start()
```

  - A função `start()` é executada apenas uma vez quando o Component do script é criado.
  - Geralmente é usada para inicializar variáveis, definir configurações, etc.

Funções de Evento

- Outra **função de evento** muito utilizada na Unity é a função `Update()`. Ela é executada continuamente enquanto o Component do script estiver ativo.

```
using UnityEngine;
using System.Collections;

public class MeuScript : MonoBehaviour{

    void Start ()
    {
        Debug.Log("Hello World!");
    }

    void Update ()
    {
        Debug.Log("I am alive!");
    }
}
```

Variáveis – Tipos de Dados

- Principais tipos de dados primitivos da linguagem C#:

Tipo	Exemplos de Valores
`int`	0, 1, 50, 200
`float`	1.2, 58.9, 0.005
`double`	2.584411, 0.000564813
`bool`	true, false
`char`	‘a’, ‘b’, ‘c’
`string`	“oi”, “tchau”, “teste”
`object`	Tipo base

Outros tipos primitivos: https://msdn.microsoft.com/en-us/library/ms228360(v=vs.90).aspx

Declaração de Variáveis em C#

- Variáveis devem ser explicitamente declaradas
- Variáveis podem ser declaradas em conjunto

Exemplos:

```
int a;          // declara uma variável chamada a do tipo int
float b;        // declara uma variável chamada b do tipo float
int d, e;       // declara duas variáveis do tipo int
double d = 0.8;   // declaração e inicialização da variável
string f = "bla"; // declaração e inicialização da variável
```

Operadores Aritméticos

- **Operadores aritméticos** são usados para se realizar operações aritméticas com as variáveis e constantes.

Operação	**Símbolo**
Adição	+
Subtração	-
Multiplicação	*
Divisão	/
Resto da Divisão	%

Exemplos:

operador de atribuição

```
total = preco * quantidade;
media = (nota2 + nota2)/2;
resultado = 3 * (1 - 2) + 4 * 2;
resto = 4 % 2;
```

Criando Novas Funções

```
tipo_de_retorno nome_da_funcao (parametros)
{
  variaveis locais

  instrucoes em C#
}
```

Se uma função não tem retorno colocamos *void*.

Se uma função não tem uma lista de parâmetros colocamos apenas o ().

Consiste no bloco de comandos que compõem a função.

Criando Novas Funções

```
using UnityEngine;
using System.Collections;

public class MeuScript : MonoBehaviour {

  float celsius_fahrenheit(float tc)
  {
    float f;
    f = 1.8f * tc + 32;
    return f;
  }

  void Start()
  {
    Debug.Log("Temperatura: " + celsius_fahrenheit(25.5f));
  }
}
```

Exercício 10

- Os alunos do IPRJ estão desenvolvendo um jogo de corrida e precisam calcular o número mínimo de litros de combustível que devem ser colocados no tanque de um carro virtual para percorrer uma pista por um determinado número de voltas até o primeiro reabastecimento. Você deve escrever uma função em C# que receba: o comprimento da pista (em metros), o número total de voltas a serem percorridas, o número de reabastecimentos desejados e o consumo de combustível do carro (em Km/L). A sua função deve calcular e retornar o número mínimo de litros necessários para percorrer até o primeiro reabastecimento.
  - OBS: considere que o número de voltas entre os reabastecimentos é o mesmo.

Introdução à Orientação a Objetos

- Consiste em um paradigma de **análise**, **projeto** e **programação** de sistemas baseado na composição e interação entre diversas unidades de software chamadas de **objetos**.

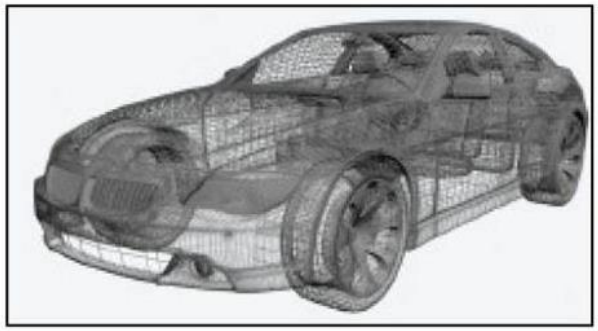
- Sugere a diminuição da distância entre a **modelagem computacional** e o **mundo real**:
  - O ser humano se relaciona com o mundo através de conceitos de objetos;
  - Estamos sempre identificando qualquer objeto ao nosso redor;
  - Para isso lhe damos nomes, e de acordo com suas características lhes classificamos em grupos;

Classes e Objetos

- A classe é o **modelo** ou **molde** de construção de objetos. Ela define as características e comportamentos que os objetos irão possuir.

- E sob o ponto de vista da programação:
  - O que é uma classe?
  - O que é um atributo?
  - O que é um método?
  - O que é um objeto?
  - Como o objeto é usado?
  - Como tudo isso é codificado???

Classe Carro



Objeto Carro2

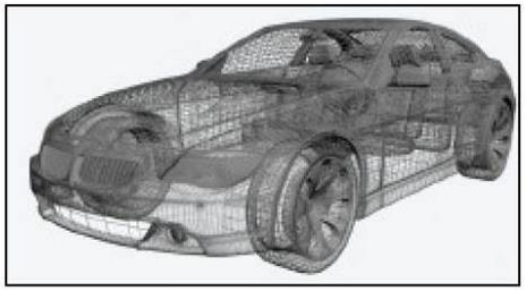
Criação de Classes

```
public class Carro {

    private string marca;
    private string cor;
    private string placa;
    private int portas;
    private int marcha;
    private double velocidade;

    public void Acelerar()
    {
        velocidade += marcha * 10;
    }

    public void Frear()
    {
        velocidade -= marcha * 10;
    }

    ...
}
```

Declaração

Atributos

Métodos



Carro
- Marca: Texto - Cor: Texto - Placa: Texto - N° Portas: Inteiro ...
+ Acelerar(): void + Frear(): void + TrocarMarcha(x): void + Buzinar(): void ...

Utilizando Objetos

- Para manipularmos objetos precisamos declarar **variáveis de objetos.**
- Uma variável de objeto é uma **referência** para um objeto.
- A **declaração de uma variável de objeto** é semelhante a declaração de uma variável normal:

```
Carro carro1;     /* carro1 não referencia nenhum objeto,
                     o seu valor inicial é null */
```

- A simples declaração de uma variável de objeto não é suficiente para a **criação de um objeto**.

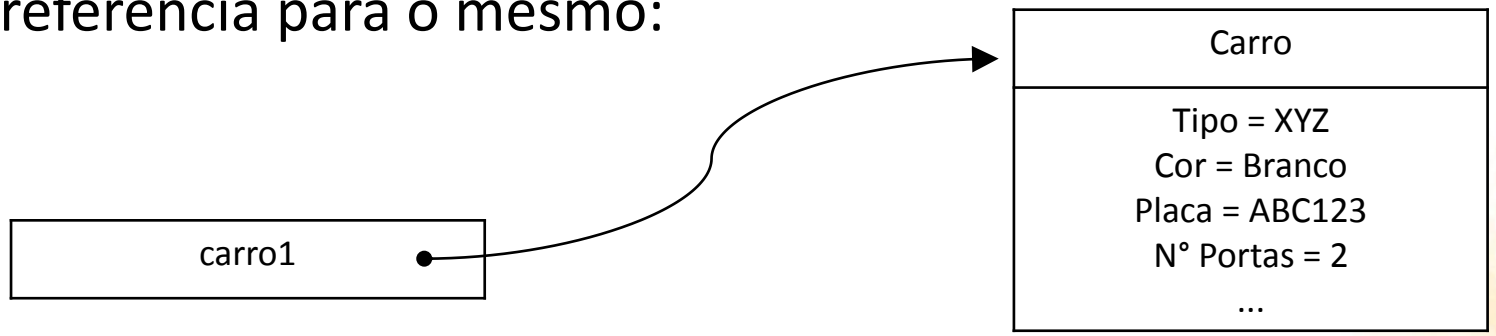
Utilizando Objetos

- A criação de um objeto deve ser explicitamente feita através do operador **new**:

```
Carro carro1;

carro1 = new Carro(); /* instancia o objeto carro1 */
```

Método construtor da classe Carro.

- O operador new **aloca o objeto em memória** e retorna uma referência para o mesmo:



Utilizando Objetos

- Um objeto expõe o seu **comportamento** através dos métodos.

- É possivel **enviar mensagens** para os objetos objetos instanciados:

```
Carro carro1 = new Carro();
Carro carro2 = new Carro();

carro1.mudarMarcha(2);
carro1.aumentaVelocidade(5);

carro2.mudarMarcha(4);
carro2.aumentaVelocidade(10);
```

Envia a mensagem mudarMarcha para o objeto carro1.

Envia a mensagem aumentaVelocidade para o objeto carro2.

Exercício 11

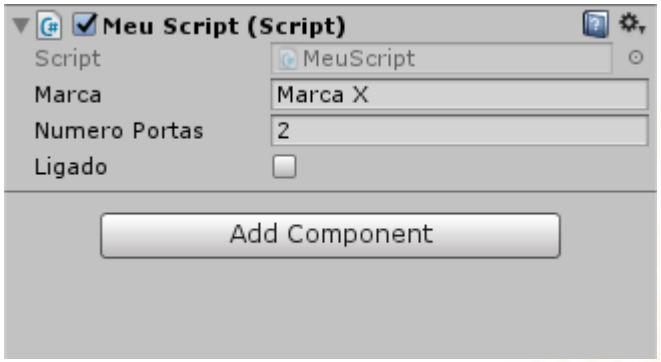
- Uma lâmpada moderna possui as seguintes características:
  - A lâmpada possui 2 estados: ligada e desligada;
  - A lâmpada pode ser configurada para emitir qualquer cor dada no formato RGB (red, green, blue);
  - A intensidade da lâmpada pode ser ajustada com qualquer valor entre 1% e 100%;
  - A lâmpada tem acesso a internet e pode tocar qualquer música que for solicitada.
- Crie uma classe para representar a lâmpada moderna, incluindo todos os atributos e métodos que possibilitem o seu funcionamento.

Classes e Components

- Os components de um GameObject são **instancias de classes**.

- Ao associar um script a um GameObject, os valores de seus **atributos públicos** podem ser editados pelo Inspector.

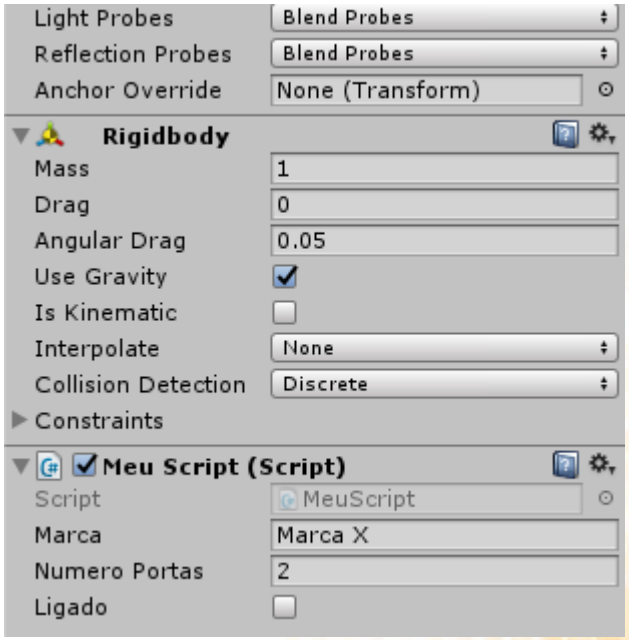
```
using UnityEngine;
using System.Collections;

public class MeuScript : MonoBehaviour {

    public string marca = "Marca X";
    public int numeroPortas = 2;
    public bool ligado = false;
    private double velocidade = 80;

    ...
}
```



Acessando Outros Components

- É comum que um script/component precise **acessar outros components** associados a um mesmo GameObject.

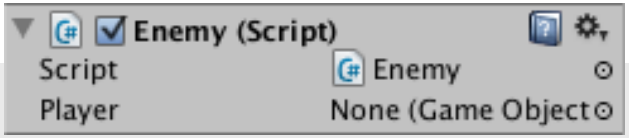
```
using UnityEngine;
using System.Collections;

public class MeuScript : MonoBehaviour {

  void Start ()
  {
    Rigidbody rb = GetComponent<Rigidbody>();

    rb.mass = 2;
    rb.AddForce(Vector3.right * 100f);
  }
}
```



Acessando Outros Objetos

- É comum que um script/component precise **acessar outros objetos**. Tal acesso pode ser realizado através de links usando variáveis ou através de buscas por nome ou tag.

- **Link através de variáveis:**




```
public class Enemy : MonoBehaviour {

  public GameObject player;

  void Start()
  {
    transform.position = player.transform.position - Vector3.forward * 10f;
  }
}
```

Acessando Outros Objetos

- **Busca por nome:**

```
public class Enemy : MonoBehaviour {
  public GameObject player;

  void Start(){
    player = GameObject.Find("MainHeroCharacter");
  }
}
```

- **Busca por tag:**

```
public class Enemy : MonoBehaviour {
  GameObject player;
  GameObject[] enemies;

  void Start() {
    player = GameObject.FindWithTag("Player");
    enemies = GameObject.FindGameObjectsWithTag("Enemy");
  }
}
```

Funções de Evento

- Além das funções de evento `Start` e `Update`, a Unity fornece um grande conjunto funções representando outros eventos de gameplay.

- Essas funções são herdadas das classes `MonoBehaviour`:

http://docs.unity3d.com/ScriptReference/MonoBehaviour.html

OnCollisionEnter	OnCollisionEnter is called when this collider/rigidbody has begun touching another rigidbody/collider.
OnCollisionEnter2D	Sent when an incoming collider makes contact with this object's collider (2D physics only).
OnCollisionExit	OnCollisionExit is called when this collider/rigidbody has stopped touching another rigidbody/collider.
OnCollisionExit2D	Sent when a collider on another object stops touching this object's collider (2D physics only).
OnCollisionStay	OnCollisionStay is called once per frame for every collider/rigidbody that is touching rigidbody/collider.
OnCollisionStay2D	Sent each frame where a collider on another object is touching this object's collider (2D physics only).

Funções de Evento

- **Update():** é executado antes da geração de cada frame.
  - **Atenção:** a frequência de execução da função é dependente da taxa frames gerados por segundos (FPS);

```
using UnityEngine;
using System.Collections;

public class MeuScript : MonoBehaviour {

    public float speed = 40;

    void Update()
    {
        transform.Rotate(Vector3.up * speed * Time.deltaTime);
    }
}
```

Funções de Evento

- **FixedUpdate():** é executado antes da atualização da simulação física em uma frequência fixa.
  - Deve ser utilizada sempre que for necessário manipular um RigidBody continuamente;

```
using UnityEngine;
using System.Collections;

public class MeuScript : MonoBehaviour {
    private Rigidbody rb;
    void Start()
    {
        rb = GetComponent<Rigidbody>();
    }
    void FixedUpdate()
    {
        rb.AddForce(Vector3.right * 10);
    }
}
```

Funções de Evento

- **OnGUI():** utilizado para renderizar elementos de tela (labels, botões, formulários, etc.).

```
using UnityEngine;
using System.Collections;

public class MeuScript : MonoBehaviour {

  private Rect helloPos;

  void Start()
  {
    helloPos = new Rect((Screen.width/2)-50,(Screen.height/2)-50,100,30);
  }

  void OnGUI()
  {
    GUI.Label(helloPos, "Hello World!");
  }
}
```

Exercício 12

- Crie um script para exibir na tela as informações de um determinado objeto da cena.
  - **As seguintes informações devem ser exibidas:** nome, posição, rotação, escala e a indicação se o *MeshRender* do objeto está configurado para receber sombras;
  - **O script criado deve ser associado a outro objeto da cena** (não ao mesmo objeto cujas informações serão exibidas);
  - As informações deve ser exibidas na **tela do jogo** usando um Label no evento OnGUI.

Estruturas Condicionais

- Em C#, estruturas condicionais são construída através do comando ***if***:

```
if (expressão_lógica)
{
    // bloco de comandos
}
```

- Exemplo:

```
if (vida <= 0)
{
    gameover = true;
}
```

Os comandos do **bloco de comandos** somente são executados se a **expressão lógica** for verdadeira

Estruturas Condicionais

- Também é possível usar o comando **else** para executar algo quando a expressão lógica não é verdadeira:

```
if (expressão_lógica)
{
   // Bloco de comandos
}
else
{
   // Bloco de comandos
}
```

Exemplo:

```
if (vida > 0)
{
   vida = vida - 1;
}
else
{
   gameover = true;
}
```

Estruturas Condicionais

- Também é possível criar sequencias de comandos **if-else** para a verificação exclusiva de varias condições:

```
if (condição_1){
   // Bloco de comandos 1
}
else if (condição_2){
   // Bloco de comandos 2
}
else if (condição_3){
   // Bloco de comandos 3
}
```

Se a primeira condição resultar em *verdadeiro*, apenas o primeiro bloco de comandos é executado, e as outras condições não são sequer avaliadas.
Senão, se a segunda condição resultar em *verdadeiro*, apenas o segundo bloco de comandos é executado, e assim por diante.

Expressões Booleanas

- Uma expressão booleana é construída através da utilização de **operadores relacionais:**

Descrição	Símbolo
Igual a	==
Diferente de	!=
Maior que	>
Menor que	<
Maior ou igual a	>=
Menor ou igual a	<=

Exemplos:

X = 10 e Y = 5

Expressão	Resultado
`X == Y`	Falso
`X != Y`	Verdadeiro
`X > Y`	Verdadeiro
`X < Y`	Falso
`X >= Y`	Verdadeiro
`X <= Y`	Falso

Todos estes operadores comparam **dois operandos**, resultando no valor falso ou verdadeiro.

Expressões Booleanas

- Expressões booleanas também podem ser combinadas através de **operadores lógicos**.

Operador	Significado	Símbolo em C#
Conjunção	E	&&
Disjunção	OU	||
Negação	NÃO	!

Exemplos:

Expressão	Resultado
`(X > 0) && (X == Y)`	Falso
`(X > 0) || (X == Y)`	Verdadeiro
`!(Y < 10)`	Falso

X = 10
Y = 5

Interação pelo Teclado

- A classe `Input` possui diversos métodos para interação pelo teclado, mouse e joystick.
  - http://docs.unity3d.com/ScriptReference/Input.html

- Os principais métodos para interação pelo teclado são:
  - `bool GetKey(KeyCode key)` - retorna true enquanto o jogador estiver pressionando a tecla;
  - `bool GetKeyDown(KeyCode key)` - retorna true no frame em que o jogador pressionou a tecla;
  - `bool GetKeyUp(KeyCode key)` - retorna true no frame em que o jogador soltou a tecla;

Código das teclas: http://docs.unity3d.com/ScriptReference/KeyCode.html

Interação pelo Teclado

- É necessário utilizar uma **estrutura condicional** para verificar se uma tecla foi pressionada.

Exemplo sem Física

```
public class ControleTeclado : MonoBehaviour {

  private float moveForce = 5;

  void Update()
  {
    if (Input.GetKey(KeyCode.RightArrow))
    {
      transform.Translate(Vector3.right * Time.deltaTime * moveForce);
    }
  }
}
```

Interação pelo Teclado

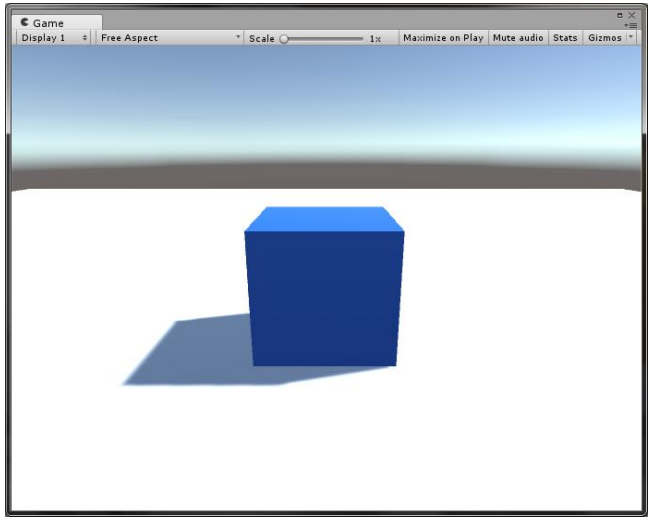
- É necessário utilizar uma **estrutura condicional** para verificar se uma tecla foi pressionada.

Exemplo com Física

```
public class ControleTeclado : MonoBehaviour {

    private Rigidbody rb;
    public float moveForce = 50;

    void Start(){
        rb = GetComponent<Rigidbody>();
    }

    void FixedUpdate(){
        if (Input.GetKey(KeyCode.RightArrow))
        {
            rb.AddForce(Vector3.right * moveForce);
        }
    }
}
```

Exercício 13

- Continue a implementação do exemplo de interação pelo teclado fazendo com que o cubo possa mover-se em 4 direções (para direita, para esquerda, para frente e para traz).
  a) Implemente a versão sem física;
  b) Implemente a versão com física;



Interação pelo Mouse

- A classe `Input` possui diversos métodos para interação pelo teclado, mouse e joystick.
  - http://docs.unity3d.com/ScriptReference/Input.html

- Os principais métodos para interação pelo mouse são:
  - `bool GetMouseButton(int button)` - retorna true enquanto o jogador estiver pressionando o botão do mouse;
  - `bool GetMouseButtonDown(int button)` - retorna true no frame em que o jogador pressionou o botão do mouse;
  - `bool GetMouseButtonUp(int button)` - retorna true no frame em que o jogador soltou o botão do mouse;
  - `Vector3 mousePosition` - representa a posição atual do mouse em coordenadas de tela;

Código dos botões: 0 – botão da direita; 1 – botão da esquerda; 2 – botão do meio.

Interação pelo Mouse

- É necessário utilizar uma **estrutura condicional** para verificar se um botão do mouse foi pressionado.

```
public class ControleMouse : MonoBehaviour {

  private float moveForce = 5;

  void Update()
  {
    if (Input.GetMouseButton(1))
    {
      transform.Translate(Vector3.right * Time.deltaTime * moveForce);
    }
  }
}
```

Criando e Destruindo GameObjects

- Embora alguns jogos mantenham uma quantidade fixa de objetos durante o jogo, muitas vezes é necessário que objetos sejam **criados e destruídos dinamicamente por scripts**.

- Método `Instantiate`:

```
public GameObject obj;

void Start ()
{
  Instantiate(obj, new Vector3(0, 5, 0), Quaternion.identity);
}
```

Criando e Destruindo GameObjects

- Embora alguns jogos mantenham uma quantidade fixa de objetos durante o jogo, muitas vezes é necessário que objetos sejam **criados e destruídos dinamicamente por scripts**.

- Método `destroy`:

```
void OnCollisionEnter(Collision obj)
{
    if (obj.gameObject.tag == "Missile")
    {
        Destroy(gameObject);
    }
}
```

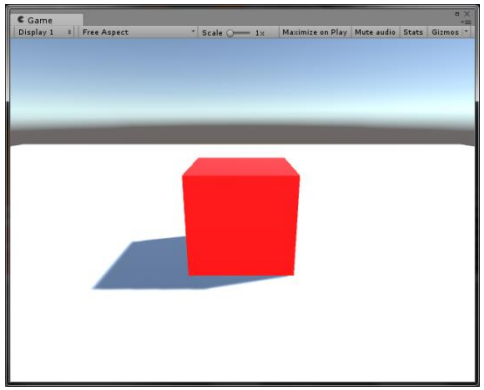
Interação pelo Mouse

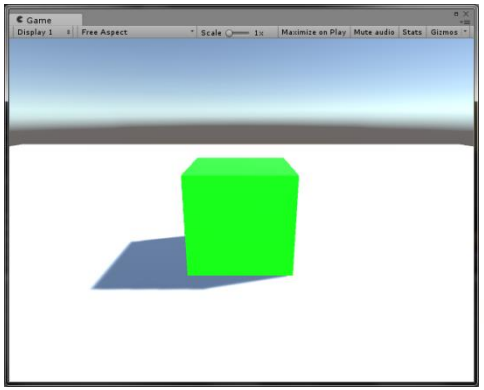
- Destruir objetos com clique do mouse:

```
public class ControleMouse : MonoBehaviour {

  void Update()
  {
    if (Input.GetMouseButtonDown(0))
    {
      Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
      RaycastHit hit;
      if (Physics.Raycast(ray, out hit, 100))
      {
        if (hit.transform.gameObject.tag == "Enemy")
        {
          Destroy(hit.transform.gameObject);
        }
      }
    }
  }
}
```

Exercício 14

- Continue a implementação do exercício de interação pelo teclado com física fazendo com que o cubo mude de cor quando o jogador clicar sobre ele.
  - O Cubo deve ficar **vermelho** se o jogador clicar nele com o botão esquerdo do mouse;
  - O Cubo deve ficar **verde** se o jogador clicar nele com o botão direito do mouse;
  - **Dica:** para alterar a cor de um objeto é necessário acessar o componente “Renderer” do objeto, o qual possui a propriedade “material.color”;





Estruturas de Repetição (`while`)

- **Estruturas de repetição** são utilizadas para indicar que um determinado conjunto de instruções deve ser executado um número definido ou indefinido de vezes, ou enquanto uma condição não for satisfeita.

- Em C#, uma das formas de se trabalhar com repetições é através do comando **while**:

```
...
while (expressão_lógica)
{
    // Bloco de comandos
}
...
```

Enquanto a "**expressão_lógica**" for verdadeira, o "bloco de comandos" é executado.

Depois, a execução procede nos comandos subsequentes ao bloco while.

Estruturas de Repetição – Exemplos

- **Exemplo 1:**

"Escrever todos os números entre 0 e 100"

- **Exemplo 2:**

"Fatorial de um número não-negativo"

$$n! = \prod_{i=1}^{n} i = n \times (n-1) \times (n-2) \times \cdots \times 3 \times 2 \times 1$$

```
void EscreveNumeros()
{
  int x = 0;
  while (x <= 100){
    Debug.Log(x);
    x++;
  }
}

int Fatorial(int n)
{
  int f = 1;
  while (n > 1){
    f = f * n;
    n = n - 1;
  }
  return f;
}
```

Estruturas de Repetição (`for`)

- Outra forma de se trabalhar com repetições é através do comando **for** – que é equivalente ao comando **while** com uma sintaxe mais compacta:

```
...

for (expressão_inicial; expressão_lógica; expressão_atualização)
{
   // Bloco de comandos
}

...
```

Estruturas de Repetição – Exemplo 1

- **Exemplo 1:** “Escrever os números entre 0 e 100”

```
void EscreveNumeros()
{
  int x = 0;
  while (x <= 100)
  {
    Debug.Log(x);
    x++;
  }
}
```

```
void EscreveNumeros()
{
  int x;
  for (x = 0; x <= 100; x++)
  {
    Debug.Log(x);
  }
}
```

Estruturas de Repetição (do-while)

- A estrutura **while** avalia a expressão booleana que controla a execução do bloco de comandos no **início do laço**.

- A linguagem C# oferece uma terceira construção de laços através do comando **do-while**:
  - A expressão booleana é avaliada no final do laço.
  - Isso significa que o bloco de comandos é executado pelo menos uma vez.

```
...
do{
   // Bloco de comandos
} while(expressão_lógica)
...
```

Estruturas de Repetição – Exemplo 1

- **Exemplo 1:** “Escrever os números entre 0 e 100”

```
void EscreveNumeros()
{
  int x = 0;
  while (x <= 100)
  {
    Debug.Log(x);
    x++;
  }
}
```

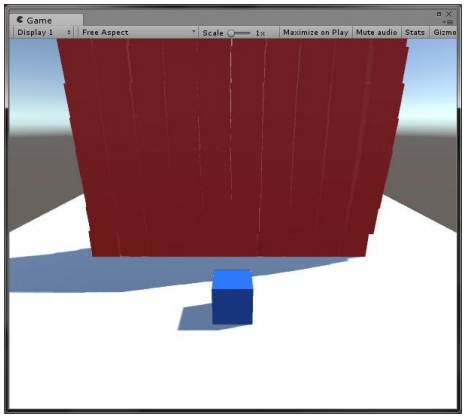
```
void EscreveNumeros()
{
  int x = 0;
  do{
    Debug.Log(x);
    x++;
  } while (x <= 100);
}
```

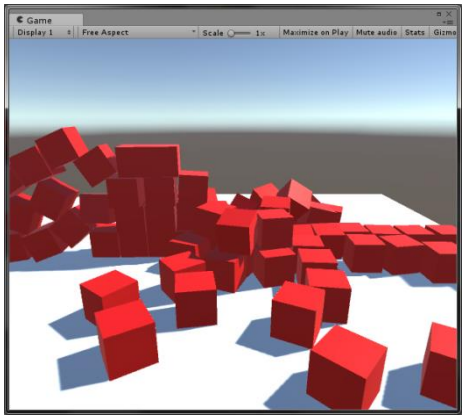
Criando Vários GameObjects

```
public class MeuScript : MonoBehaviour {

  public GameObject obj;

  void Start ()
  {
    for (int x = 0; x < 10; x++)
    {
      Instantiate(obj, new Vector3(x*0.5f, x*1.5f, 0),
                  Quaternion.identity);
    }
  }
}
```

Exercício 15

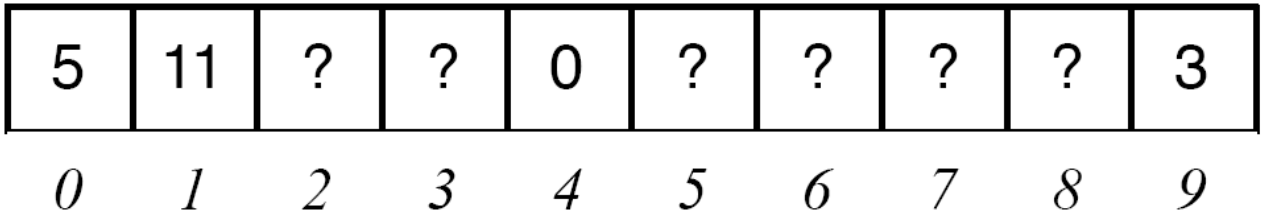
- Continue a implementação do exercício de interação pelo teclado com física criando uma parede formada por cubos com RigidBodies.
  - Lembre-se de criar um **Prefab** para o cubo usado para a parede;
  - Configure a **massa e a força** aplicada sobre o cubo controlado pelo teclado para que ele seja capaz de derrubar a parede quando lançado contra ela;





Arrays em C#

- **Array** é um mecanismo que nos permite armazenar um conjunto de valores na memória do computador.



- Em C#, arrays são objetos. Consequentemente, uma variável do tipo array é na verdade uma referência para um objeto.

```
int[] myArray;
```

Arrays – Declaração

- Ao declararmos um array em C#:

```
int[] myArray;
```

  - Nenhuma área de memória para o array é alocada, apenas uma referência para um array de inteiros é definida;
  - Nenhuma referência ao tamanho do array é feita na declaração;
  - Apenas na criação do array é que iremos alocar espaço em memória e, consequentemente, iremos definir o seu tamanho.

Arrays – Criação

- Antes de utilizar um array, ele deve ser explicitamente criado:

```
myArray = new int[10];
```

- Podemos declarar e criar um array:

```
int[] myArray = new int[10];
```

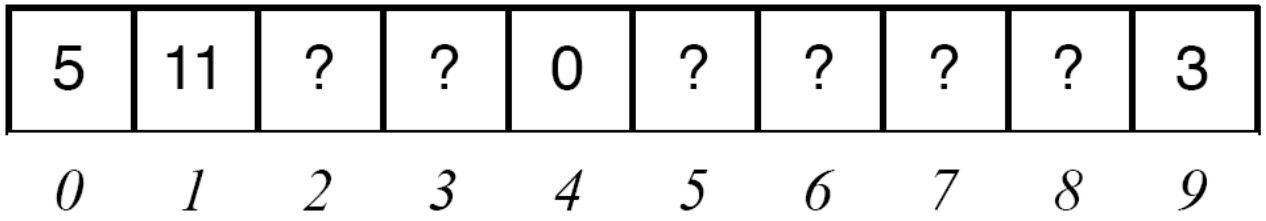
- Podemos declarar, criar e inicializar um array:

```
int[] myArray = new int[]{10, 20, 30, 40};
```

Arrays – Incialização

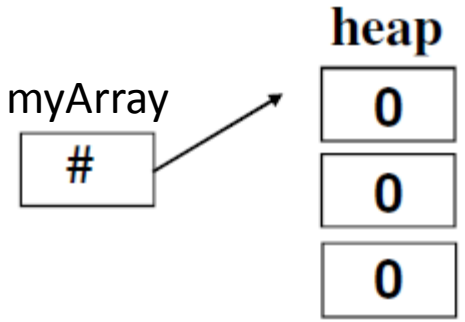
- Os elementos do vetor podem ser acessados através dos seus índices:

```
int[] myArray = new int[10];

myArray[0] = 5;
myArray[1] = 11;
myArray[4] = 0;
myArray[9] = 3;
```

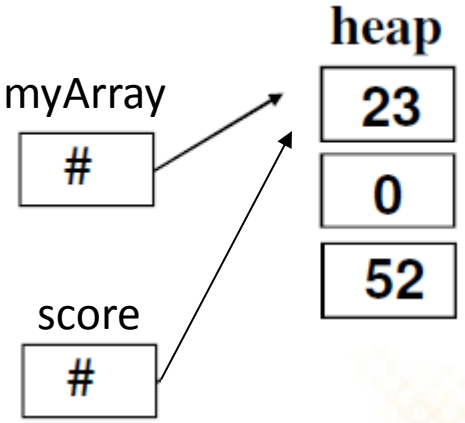


Arrays – Inicialização

```
int[] myArray = new int[3];
```

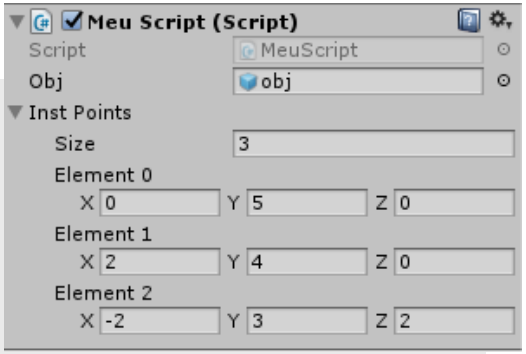



```
myArray[0] = 23;
myArray[1] = 0;
myArray[2] = 52;
```



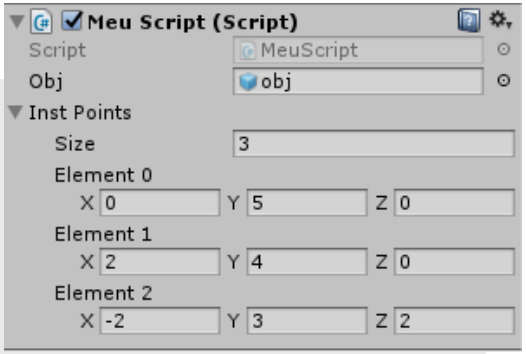

```
int[] score = myArray;
```

Array de Posições




```
public class MeuScript : MonoBehaviour {
  public GameObject obj;
  public Vector3[] instPoints;

  void Start () {
    for (int i = 0; i < instPoints.Length; i++)
    {
      Instantiate(obj, new Vector3(instPoints[i].x,
                                   instPoints[i].y,
                                   instPoints[i].z),
                                   Quaternion.identity);
    }
  }
}
```

Array de Posições




```
public class MeuScript : MonoBehaviour {

  public GameObject obj;
  public Vector3[] instPoints;

  void Start ()
  {
    foreach (Vector3 pos in instPoints)
    {
      Instantiate(obj, new Vector3(pos.x, pos.y, pos.z),
                  Quaternion.identity);
    }
  }
}
```

Estrutura de repetição `foreach`

Arrays Multidimensionais

- Em C#, arrays podem ter mais de uma dimensão:

```
int[,] mat;
```

3	1	8	6	1
7	2	5	4	9
1	9	3	1	2
5	8	6	7	3
6	4	9	2	1

Arrays Multidimensionais – Criação

- Da mesma forma que arrays simples, arrays multidimensionais também devem ser explicitamente criados:

```
int[,] mat = new int[3,3];
```

- Eles também podem ser criados e inicializados:

```
int[,] numbers = new int[3, 2]{{1, 2},
                               {3, 4},
                               {5, 6}};
```

Arrays Multidimensionais

- Os elementos do array multidimensional podem ser acessados através dos seus índices:

```
int[,] mat = new int[3,3];

mat[0,0] = 5;
mat[0,2] = 1;
mat[2,1] = 8;
```

5	?	1
?	?	?
?	8	?

Array Multidimensionais de GameObjects

```
public class MeuScript : MonoBehaviour {
  public GameObject obj;
  public GameObject[,] refObjects;

  void Start (){
    refObjects = new GameObject[10, 10];
    for (int x = 0; x < 10; x++){
      for (int y = 0; y < 10; y++){
        refObjects[x, y] = (GameObject)Instantiate(obj,
                                        new Vector3(x, 0, y),
                                        Quaternion.identity);
      }
    }
  }
```

Array Multidimensionais de GameObjects

```
  void Update()
  {
    for (int x = 0; x < 10; x++)
    {
      for (int y = 0; y < 10; y++)
      {
        refObjects[x, y].transform.Rotate(Vector3.up *
                                          50 * Time.deltaTime);
      }
    }
}
```

Estruturas de Dados Dinâmicas

- **Arrays:**
  - Ocupam um espaço contínuo de memória;
  - Permite acesso randômico;
  - Requer pré-dimensionamento de espaço de memória;

- **Estruturas de Dados Dinâmicas:**
  - Crescem (ou decrescem) à medida que elementos são inseridos (ou removidos);
  - Exemplo: listas, pilhas, filas, dicionários...

Listas

- Em C#, uma lista é um objeto que pode armazenar um conjunto de variáveis de um determinado tipo especificado de forma genérica.

```
List<T> lista = new List<T>();
```

- Exemplo:

```
List<int> numbers = new List<int>();
numbers.Add(1);
numbers.Add(2);
numbers.Add(3);
```

Listas

```
List<string> frutas = new List<string>();
```

- Método Add:

```
frutas.Add("maça");
frutas.Add("banana");
frutas.Add("laranja");
```

- Método Remove:

```
frutas.Remove("banana");
```

- Método RemoveAt:

```
frutas.RemoveAt(1);
```

Listas

```
List<string> frutas = new List<string>();
```

- Método Clear:

```
frutas.Clear()
```

- Método Contains:

```
if (frutas.Contains("banana")){ ... }
```

- Método IndexOf:

```
int pos = frutas.IndexOf("banana")
```

Exemplo – Lista (List)

```
using UnityEngine;
using System.Collections.Generic;

public class MeuScript : MonoBehaviour {

  public GameObject tilePrefab;
  private List<GameObject> selectedObjects;

  void Start ()
  {
    selectedObjects = new List<GameObject>();
    for (int x = 0; x < 10; x++){
      for (int y = 0; y < 10; y++){
        Instantiate(tilePrefab, new Vector3(x, 0, y),
                                                Quaternion.identity);
      }
    }
  }
```

Exemplo – Lista (List)

```
  void Update(){

    if (Input.GetMouseButtonDown(0)){
      Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
      RaycastHit hit;

      if (Physics.Raycast(ray, out hit, 100)){
        if (!selectedObjects.Contains(hit.transform.gameObject)){
          selectedObjects.Add(hit.transform.gameObject);
        }
      }
    }

    foreach (GameObject obj in selectedObjects){
      obj.transform.Rotate(Vector3.up * 50 * Time.deltaTime);
    }
  }
}
```

Filas – Queue

- Em C#, uma fila pode armazenar dados de um determinado tipo especificado de forma genérica. O primeiro dado a ser inserido, é o primeiro a ser removido.

```
Queue<T> fila = new Queue<T>();
```

- Exemplo:

```
Queue<int> numbers = new Queue<int>();
numbers.Enqueue(1);
numbers.Enqueue(2);
numbers.Enqueue(3);
```

Filas

```
Queue<string> frutas = new Queue<string>();
```

- Método Add:

```
frutas.Enqueue("maça");
frutas.Enqueue("banana");
frutas.Enqueue("laranja");
```

- Método Dequeue:

```
frutas.Dequeue();
```

- Método Peek:

```
string primeia = frutas.Peek();
```

Exemplo – Fila (Queue)

```
using UnityEngine;
using System.Collections.Generic;

public class MeuScript : MonoBehaviour {

  public GameObject tilePrefab;
  private Queue<Vector3> path;

  void Start ()
  {
    path = new Queue<Vector3>();
    for (int x = 0; x < 10; x++){
      for (int y = 0; y < 10; y++){
        Instantiate(tilePrefab, new Vector3(x, 0, y),
                                                Quaternion.identity);
      }
    }
  }
```

Exemplo – Fila (Queue)

```
  void Update(){
    if (Input.GetMouseButtonDown(0)){
      Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
      RaycastHit hit;
      if (Physics.Raycast(ray, out hit, 100)){
        path.Enqueue(hit.transform.position);
      }
    }
    if (path.Count > 0){
      if (transform.position != path.Peek()){
        transform.position = Vector3.MoveTowards(transform.position,
                             path.Peek(), 1.5f * Time.deltaTime);
      }else{
        path.Dequeue();
      }
    }
  }
}
```

Operações com Vectors

- Distância entre dois pontos:

```
float dist = Vector3.Distance(other.position, transform.position);
```

- Ângulo de visão:

```
public class MeuScript : MonoBehaviour {
  public Transform target;

  void Update() {
    Vector3 targetDir = target.position - transform.position;
    float angle = Vector3.Angle(targetDir, transform.forward);

    if (angle < 30.0f)
      Debug.Log("I can see you!");
  }
}
```

Operações com Vectors

- Mover em direção:

```
public class MeuScript : MonoBehaviour
{
  public Transform target;
  public float speed = 2;

  void Update ()
  {
    transform.position = Vector3.MoveTowards(transform.position,
                         target.position, speed * Time.deltaTime);
  }
}
```

Operações com Vectors

- Rotacionar em direção:

```
public class MeuScript : MonoBehaviour
{
  public Transform target;
  public float speed = 1;

  void Update()
  {
    Vector3 targetDir = target.position - transform.position;
    Vector3 newDir = Vector3.RotateTowards(transform.forward,
                         targetDir, speed * Time.deltaTime, 0.0f);
    transform.rotation = Quaternion.LookRotation(newDir);
  }
}
```

Pilares da Orientação a Objetos

- A orientação a objetos está sedimentada sobre quatro pilares derivados do princípio da abstração:
  - Encapsulamento
  - Herança
  - Composição
  - Polimorfismo

Pilares da Orientação a Objetos

- A orientação a objetos está sedimentada sobre quatro pilares derivados do princípio da abstração:
  - **Encapsulamento**
  - Herança
  - Composição
  - Polimorfismo

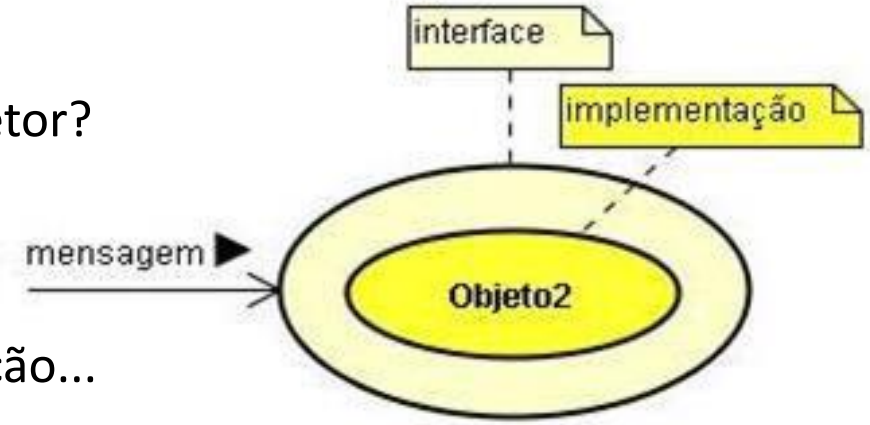
Encapsulamento

- Encapsulamento é a característica da orientação a objetos capaz de **ocultar** partes (dados e detalhes) de implementação interna de classes do mundo exterior.

- Os objetos ficam **protegidos** em uma capsula (interface).

- Evita o acesso indevido e a corrupção de dados.
  - Exemplo: como desligar o projetor?

- Vantagens:
  - Abstração, manutenção, proteção...



Modificadores de Acesso

```
public class Carro {
    public double velocidade;

    private String cor;

    public void Acelerar()
    {
        velocidade += marcha * 10;
    }

    public void Frear()
    {
        velocidade -= marcha * 10;
    }
}
```

O modificador **public** declara que a classe pode ser usada por qualquer outra classe. Sem public, uma classe pode ser usada somente por classes do mesmo namespace.

O modificador **public** declara que o atributo pode ser acessado por métodos externos à classe na qual ele foi definido.

O modificador **private** declara que o atributo não pode ser acessado por métodos externos à classe na qual ele foi definido.

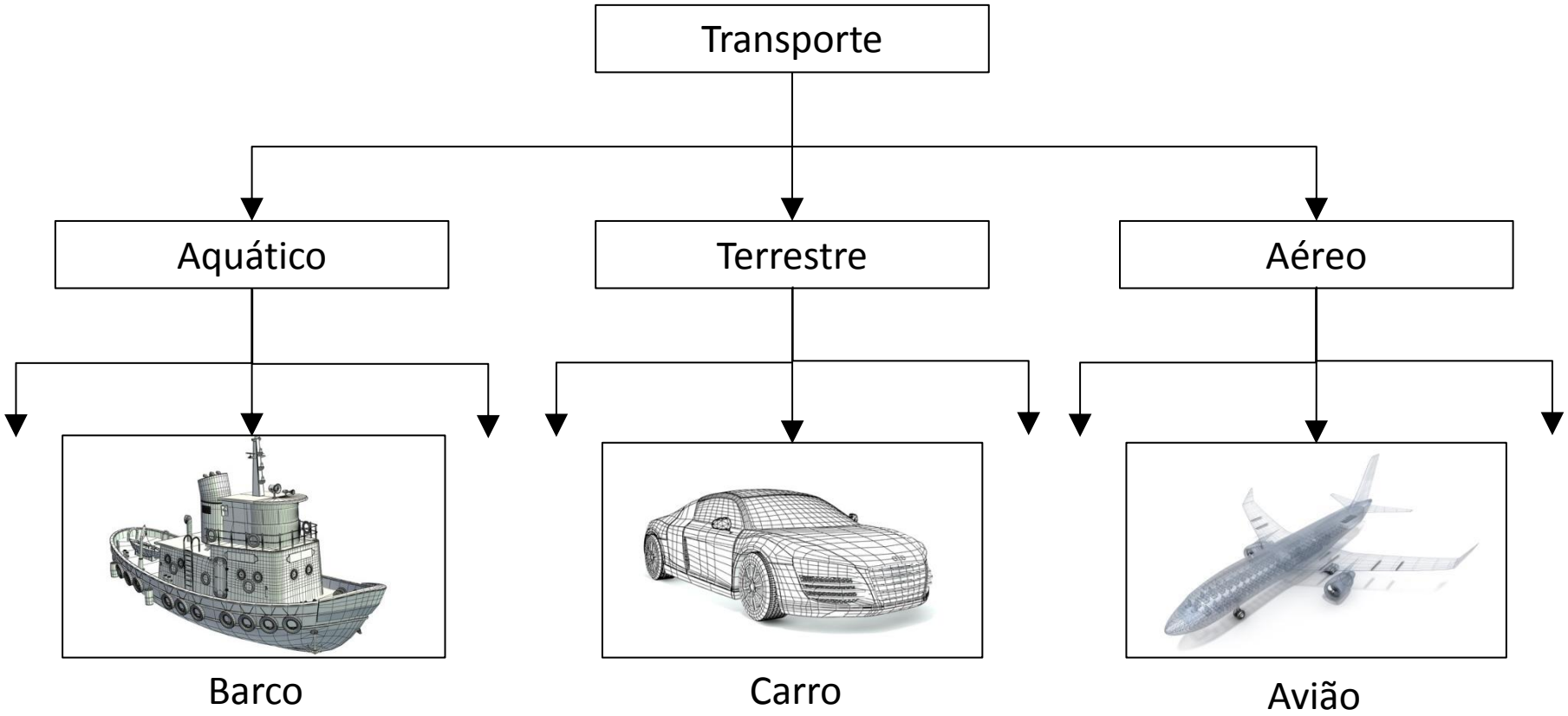
O modificador **public** declara que o método pode ser executado por métodos externos à classe na qual ele foi definido.

Pilares da Orientação a Objetos

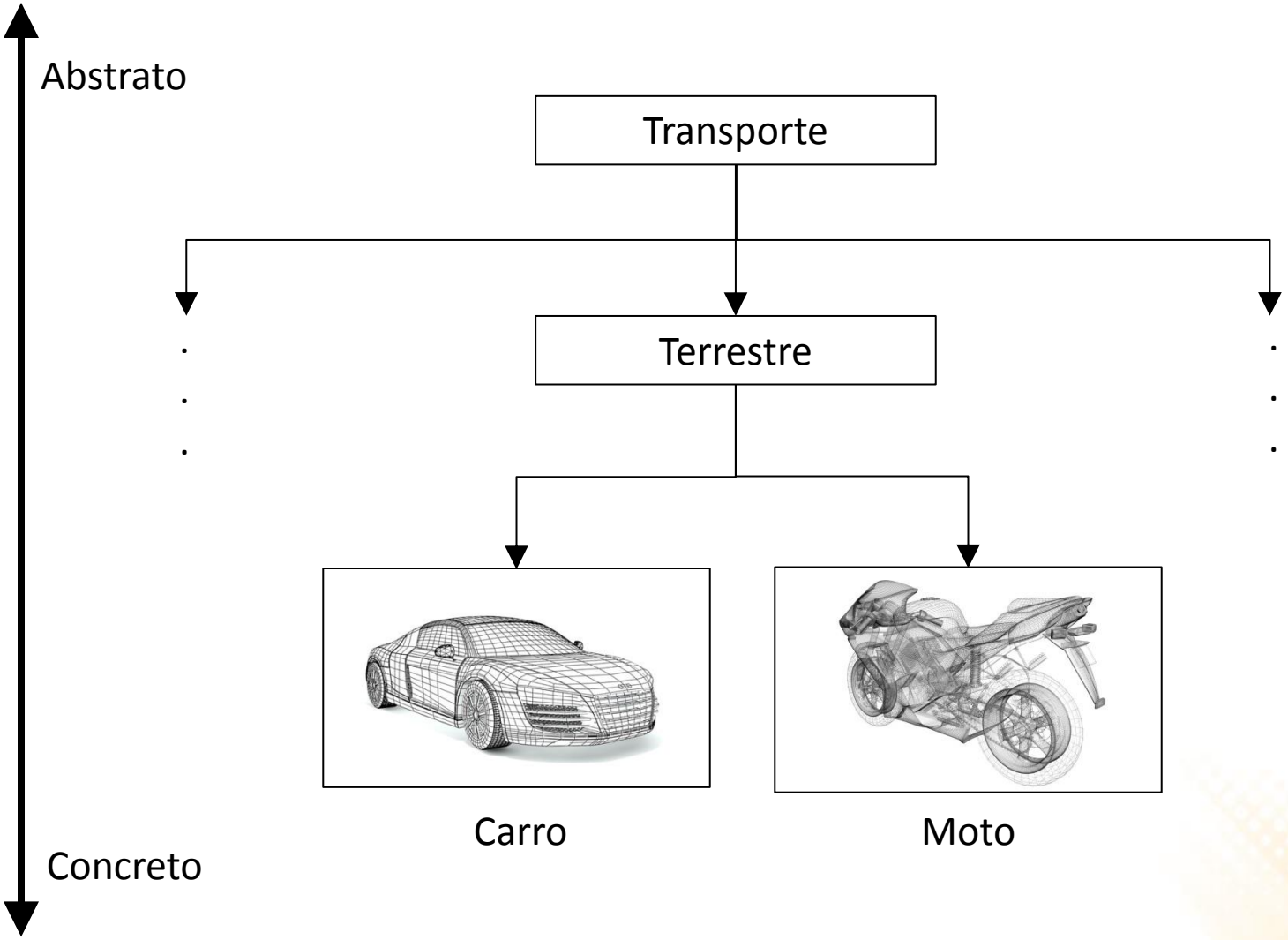
- A orientação a objetos está sedimentada sobre quatro pilares derivados do princípio da abstração:
  - Encapsulamento
  - **Herança**
  - Composição
  - Polimorfismo

Herança

- Herança é o mecanismo pelo qual uma classe pode "**herdar**" as características e métodos de outra classe para expandi-la ou especializá-la de alguma forma.



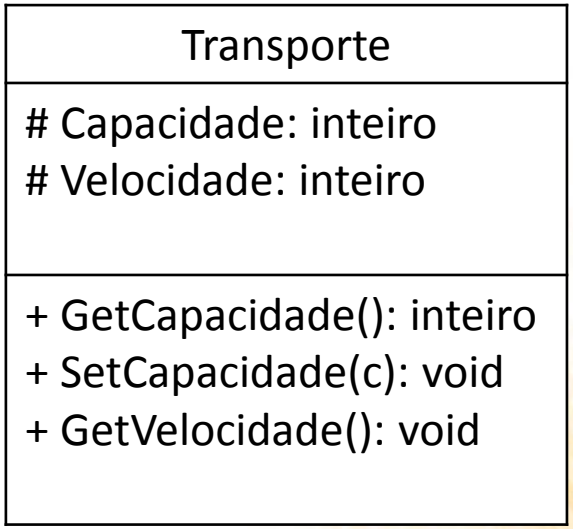
Generalização e Especialização



Herança

```
public class Transporte {
    protected int capacidade;
    protected int velocidade;

    public Transporte()
    {
        velocidade = 0;
    }
    public int GetCapacidade()
    {
        return capacidade;
    }
    public void SetCapacidade(int cap)
    {
        capacidade = cap;
    }
    public int GetVelocidade()
    {
        return velocidade;
    }
}
```

O modificador **protected** determina que apenas a própria classe e as classes filhas poderão ter acesso ao atributo.



Herança

```
public class Terrestre : Transporte{

    protected int numRodas;

    public Terrestre(int rodas)
    {
        numRodas = rodas;
    }

    public int GetNumRodas()
    {
         return numRodas;
    }

    public void SetNumRodas(int num)
    {
         numRodas = num;
    }
}
```

A classe Terrestre **herda** as características da classe Transporte.

Terrestre
N° Rodas: inteiro
+ GetNumRodas() : inteiro + SetNumRodas(n): void

Herança

```
public class Carro : Terrestre{
  private String cor;
  private String placa;
  private int marcha;

  public Carro(String ncor, String nplaca): base(4)
  {
    cor = ncor;
    placa = nplaca;
    marcha = 0;
  }
  public int GetMarcha()
  {
    return marcha;
  }
  public void TrocarMarcha(int novaMarcha)
  {
    marcha = novaMarcha;
  }
  ...
```

A classe Carro **herda** as características da classe Terrestre.

Chamada ao **método construtor** da classe pai (Terrestre).

Carro
- Cor: Texto - Placa: Texto - Marcha: Inteiro
+ GetMarcha(): Inteiro + TrocaMarcha(n): void + Acelerar(): void + Frear(): void

Herança

```
  ...


  public void Acelerar()
  {
    velocidade += marcha * 10;
  }

  public void Frear()
  {
    velocidade -= marcha * 10;
  }

}
```

Carro
- Cor: Texto - Placa: Texto - Marcha: Inteiro
+ GetMarcha(): Inteiro + TrocaMarcha(n): void + Acelerar(): void + Frear(): void

Herança

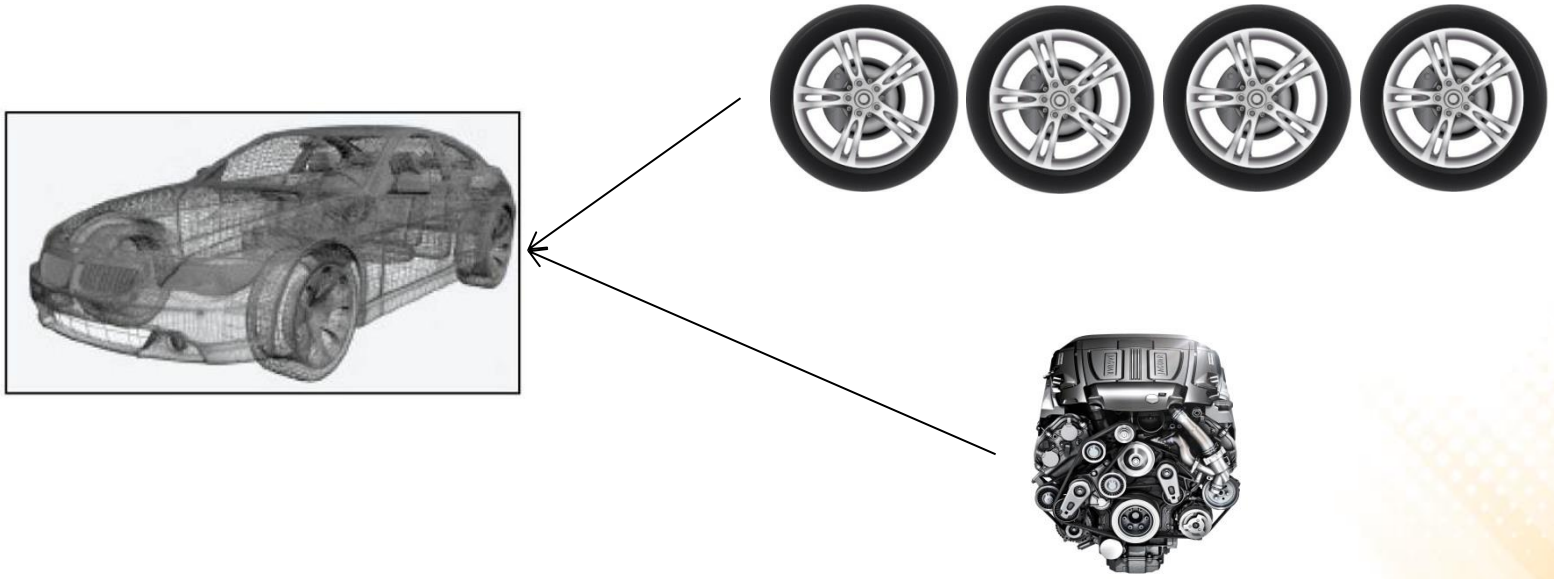
```
using UnityEngine;
using System.Collections;

public class MeuScript : MonoBehaviour {

  void Start()
  {
    Carro meuCarro = new Carro("Vermelho", "ABC-1234");

    meuCarro.TrocarMarcha(1);
    meuCarro.Acelerar();
    meuCarro.Acelerar();

    Debug.Log("Velocidade do Carro: " + meuCarro.GetVelocidade());

  }
}
```

Pilares da Orientação a Objetos

- A orientação a objetos está sedimentada sobre quatro pilares derivados do princípio da abstração:
  - Encapsulamento
  - Herança
  - **Composição**
  - Polimorfismo

Composição

- A composição (ou agregação) é a característica da orientação a objetos que permite combinar objetos simples em objetos mais complexos.
  - Possibilita a reutilização de código;
  - Um objeto mais complexo pode ser composto de partes mais simples;

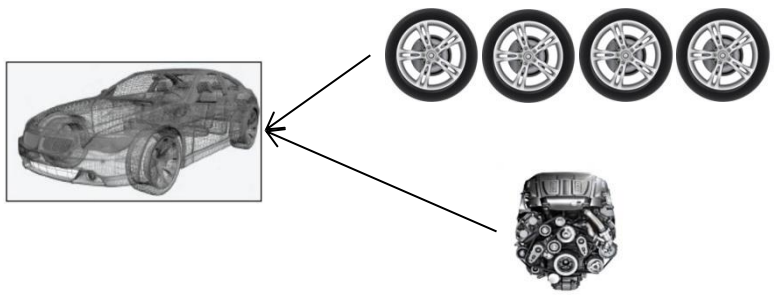


Composição

```
public class Carro{
    private Roda[] rodas;
    private Motor motor;

    public Carro()
    {
        rodas = new Roda[4];
        for (int x = 0; x < 4; x++)
           rodas[x] = new Roda();

        motor = new Motor();
    }

    ...
}
```

```
public class Roda{
    ...
    public Roda()
    {
        ...
    }
    ...
}
```

```
public class Motor{
    ...
    public Roda()
    {
        ...
    }
    ...
}
```



Pilares da Orientação a Objetos

- A orientação a objetos está sedimentada sobre quatro pilares derivados do princípio da abstração:
  - Encapsulamento
  - Herança
  - Composição
  - **Polimorfismo**

Polimorfismo

- O polimorfismo deriva da palavra polimorfo, que significa multiforme, ou que pode variar a forma.

- Na orientação a objetos, polimorfismo é a habilidade de objetos de classes diferentes responderem a mesma mensagem de diferentes maneiras.
  - Ou seja, várias formas de responder à mesma mensagem.

Polimorfismo

- O Polimorfismo pode ser classificado de três maneiras:
  - **Polimorfismo de sobrecarga:** permite que uma classe possa ter vários métodos com o mesmo nome, mas com assinaturas distintas;
  - **Polimorfismo de Sobreposição:** permite que uma classe possua um método com a mesma assinatura (nome, tipo e ordem dos parâmetros) que um método da sua superclasse (o método da classe derivada sobrescreve o método da superclasse);
  - **Polimorfismo de Inclusão:** permite que um objeto de uma classe superior assuma a forma de qualquer um dos seus descendentes;

Polimorfismo de Sobrecarga

- Polimorfismo de sobrecarga:

```
public class Carro : Terrestre{

    ...

    public void Acelerar()
    {
        velocidade += marcha * 10;
    }

    public void Acelerar(double forca)
    {
        velocidade += marcha * 10 * forca;
    }

    ...
}
```

Polimorfismo de Sobrecarga

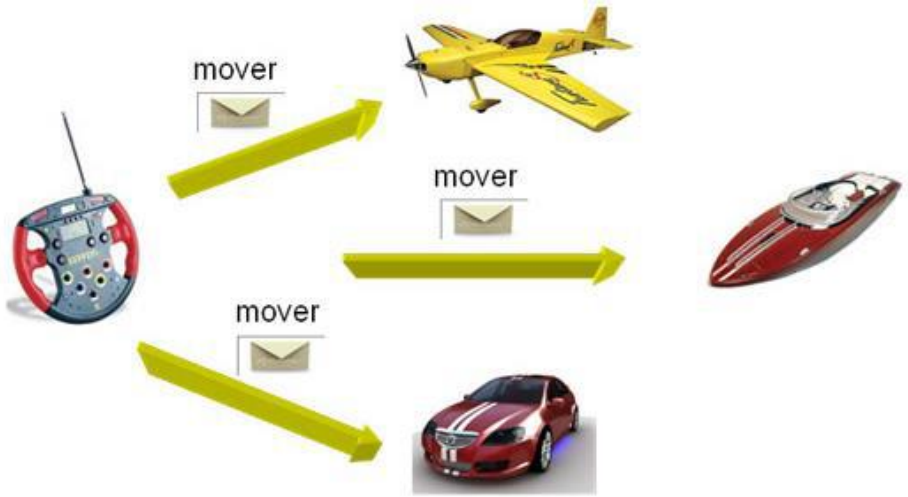
- Polimorfismo de sobrecarga de métodos construtores:

```
public class Carro : Terrestre{

    ...

    public Carro()
    {
        super(4);
    }

    public Carro(String ncor, String nplaca)
    {
        super(4);
        cor = ncor;
        placa = nplaca;
        marcha = 0;
    }
    ...
}
```

Polimorfismo de Sobreposição

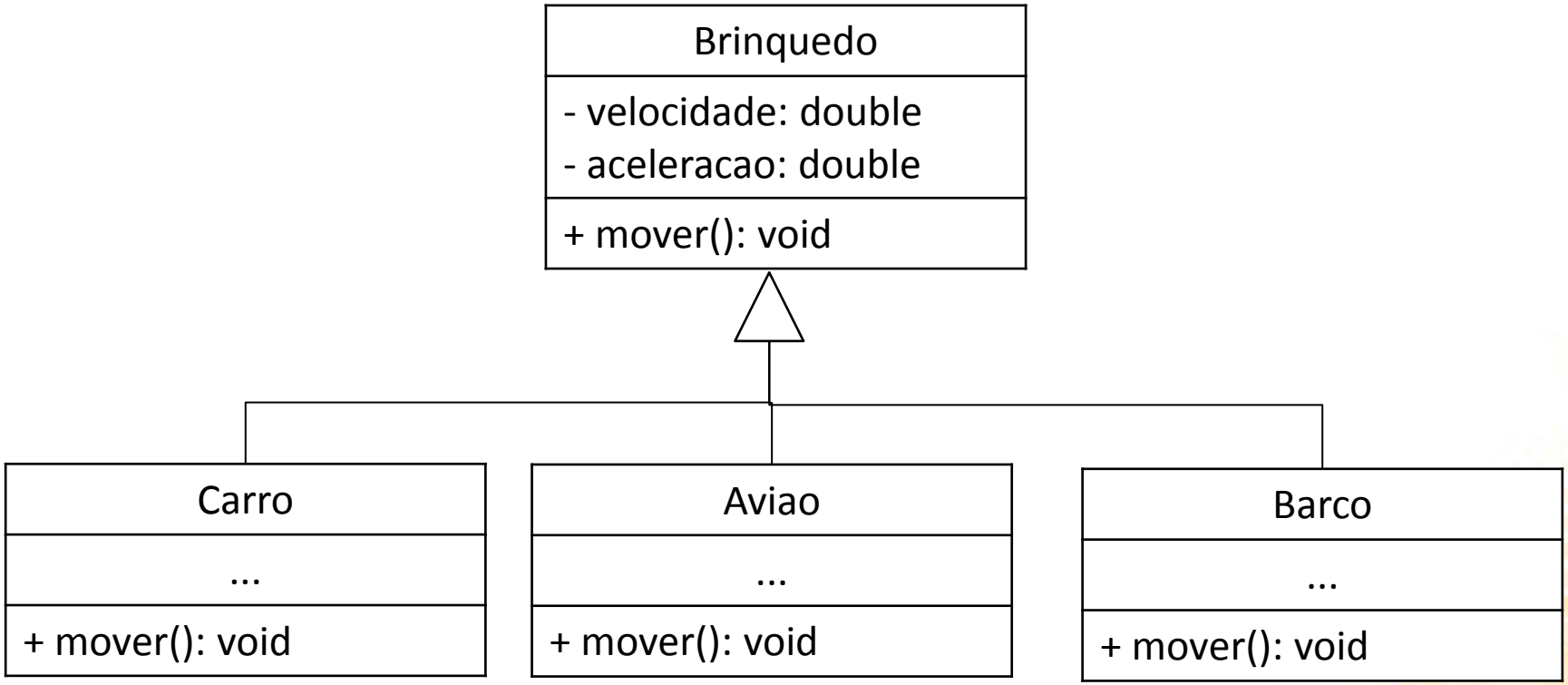
- **Exemplo:** Um dono de uma fábrica de brinquedos solicitou que seus engenheiros criassem um mesmo controle remoto para todos os brinquedos de sua fábrica.

- Assim quando o brinquedo recebe o sinal MOVER, ele se move de acordo com a sua função:
  - Para o avião, mover significa VOAR;
  - Para o barco significa NAVEGAR;
  - Para o automóvel CORRER;



Polimorfismo de Sobreposição

- Considere que a classe Brinquedo possui como descendentes as classes Carro, Avião e Barco:



Polimorfismo de Sobreposição

```
public class Brinquedo{
    ...
    public Brinquedo()
    {
    }
    public virtual void mover()
    {
        System.out.print("Mover brinquedo!");
    }
}
```

```
public class Carro : Brinquedo{
    public Carro()
    {
    }

    public override void mover()
    {
        System.out.print("CORRER!");
    }
}
```

Polimorfismo de Sobreposição

```
public class Aviao : Brinquedo{
    public Aviao()
    {
    }

    public override void mover()
    {
        System.out.print("VOAR!");
    }
}
```

```
public class Barco : Brinquedo{
    public Barco()
    {
    }

    public override void mover()
    {
        System.out.print("NAVEGAR!");
    }
}
```

Polimorfismo de Sobreposição

```
public class ControleRemoto{
    private Brinquedo brinquedo;

    public ControleRemoto(Brinquedo b)
    {
        brinquedo = b;
    }

    public void mover()
    {
        brinquedo.mover();
    }
}
```

Polimorfismo de inclusão

```
void Start(){
    Carro carro = new Carro();
    ControleRemoto = new ControleRemoto(carro);
    ControleRemoto.mover();
}
```

Material Adicional

- Scripting Reference: https://docs.unity3d.com/ScriptReference/index.html
- Scripting Manual: https://docs.unity3d.com/Manual/ScriptingSection.html
- Scripting Tutorials: https://unity3d.com/pt/learn/tutorials/topics/scripting
- Unity Forum: https://forum.unity3d.com/
- Unity Answers: http://answers.unity3d.com/