# Linguagens e Ambientes de Programação (2018/2019)

## Teórica 25 (05/jun/2019)

Resolução de problemas, para ajudar na preparação para o 2º teste.

Excecionalmente, esta aula vai ser na sala 127-II.

- PILHA - Considere o seguinte programa escrito em GCC, uma variante do C que suporta aninhamento de funções:

```
#include <stdio.h>

#define N_NODES  4
typedef struct Node { int value ; struct Node *next ; } Node, *List ;

void G(List l) {     // global function
    for( ; l->next != NULL ; l = l->next );
    // QQQ
}

int main(void) {      // global function
    Node nodes[N_NODES];
    void F(int i) {   // local function
        nodes[i].value = i;
        if( i == N_NODES-1 ) {
            nodes[i].next = NULL;
            G(&nodes[0]);
        }
         else {
            nodes[i].next = &nodes[i+1];
            F(i+1);
        }
    }
    F(0);
    return 0 ;
}
```

Mostre qual o estado da pilha de execução no momento em que a execução do programa atinge o ponto marcado com QQQ. Note que o array nodes ocupa 8 posições de memória. Não se esqueça dos registos de ativação das funções start e main.

Use as convenções habituais das aulas: Para efeitos da criação do registo de activação inicial, imagine que cada programa em GCC está embebido numa função sem argumentos chamada start. Depois trate todas as entidades globais do programa como sendo locais à função imaginária start. Assuma também que a primeira célula da pilha de execução é identificada como posição 0, a segunda célula da pilha de execução é identificada como posição 1, etc.

- JAVASCRIPT - **Vida.** O objetivo deste problema é definir em JavaScript um sistema de classes adequado à representação de espécies de seres vivos e de relações entre espécies. Esta pergunta é um pouco extensa, mas descobrirá que o problema se resolve escrevendo pouco código.

Vamos considerar **espécies** e **relações simbióticas**.

**Espécies**: Uma **espécie** é um tipo de ser vivo, que pode ser tanto animal como vegetal. Exemplos: **humano**, **lírio**, **cão**, **raposa**, **peixepalhaço**, **anémona**. Cada espécie é implementada usando uma classe com atributos que podem ser bastante variados. Contudo, existem dois atributos que são comuns a todas as espécies: ***name*** (o nome da espécie, uma string) e ***time*** (o tempo médio de vida dos membros dessa espécie em segundos, um inteiro).

**Relações simbióticas**: Existem diversos tipos de relações simbióticas: **mutualista** quando os dois participantes recolhem benefícios; **comensalistica** quando o primeiro participante beneficia e o segundo nem beneficia nem é prejudicado (geralmente o primeiro alimenta-se dos restos deixado pelo segundo); **parasítica** quando o primeiro beneficia e o segundo é prejudicado. Há outros tipos de relações simbióticas conhecidas e mais poderão vir a ser descobertos no futuro.

Cada **relação simbiótica** envolve duas componentes, que tanto podem ser espécies como podem ser outras relações simbióticas. Por exemplo: a relação **[peixepalhaço-anémona]** envolve só duas espécies; a relação **[humano-cão]** também só envolve duas espécies; mas a relação **[[humano-cão]-raposa]** já envolve uma relação simbiótica (caçador + cão) e uma espécie (raposa). Cada tipo de relação simbiótica é implementado usando uma classe com atributos que podem ser bastante variados. Mas existem dois atributos que são comuns a todas as relações simbióticas: ***l*** (o primeiro participante na relação) e ***r*** (o segundo participante na relação).

**Problema**

O objetivo deste problema é a definição dum sistema de classes bem fatorizado e extensível, adequado à representação de aspetos da **vida**. Escreva código compacto, bem fatorizado e extensível. No futuro vamos querer acrescentar novas espécies e novas variedades de relações simbióticas, e o programa tem de estar preparado para isso.

Programe apenas as classes abstratas do sistema. Se escrever classes concretas para exemplificar, deixe-as ficar vazias. As funções que todos as classes devem suportar são as seguintes quatro:

- `constructor(…)` – Função de inicialização com argumentos que dependem de cada classe.
- `getName()` – No caso duma espécie, é o nome específico desta, por exemplo "human". No caso duma relação simbiótica, é a concatenação dos nomes dos elementos envolvidos, separados por "-" e rodeados por parêntesis retos, por exemplo "[[human-dog]-fox]".
- `getTime()` – No caso duma espécie, é o tempo específico dela. No caso duma relação simbiótica, é o mínimo de todos os tempos envolvidos.
- `winWin()` - No caso duma espécie, o resultado é true. No caso duma relação simbiótica, o resultado só é true só se todas as relações envolvidas forem mutualistas. Defina este método nos sítios certos para poupar na escrita e favorecer a reutilização de código. Ou seja, fatorize bem.

Recomendamos que use as seguintes classes, já identificadas. Se preferir, pode ignorar o que se oferece e fazer diferente.

```
class Life { // abstrato
}

class Species extends Life { // abstrato

}

class Human extends Species {…} // concreto -- não defina
class Dog extends Species {…} // concreto -- não defina
class Fox extends Species {…} // concreto -- não defina

class Symbiosis extends Life { // abstrato
}

class Mutualism extends Symbiosis { // abstrato
}
```

```
class Commensalism extends Symbiosis { // abstrato
}

class Parasitism extends Symbiosis { // abstrato
}

class HumanDog extends Mutualism {…}  // concreto -- não defina
class HumanDogFox extends Parasitism {…}  // concreto -- não defina
```

---

#