

ENGENHARIA INFORMÁTICA E DE COMPUTADORES

Algoritmos e Estruturas de Dados

(parte 10 – Tipos de Dados Abstratos)

2º Semestre 2022/2023

Instituto Superior de Engenharia de Lisboa

Paula Graça


TIPOS DE DADOS ABSTRATOS (ADT)

- Um **tipo de dados abstrato (ADT)** é:
 - É um modelo matemático para os tipos de dados
 - O seu comportamento é definido por um conjunto de valores e de operações
 - É uma interface que define a estrutura dos dados e as operações sobre os dados

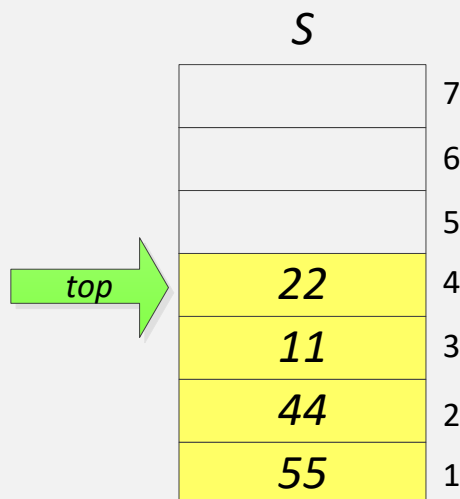
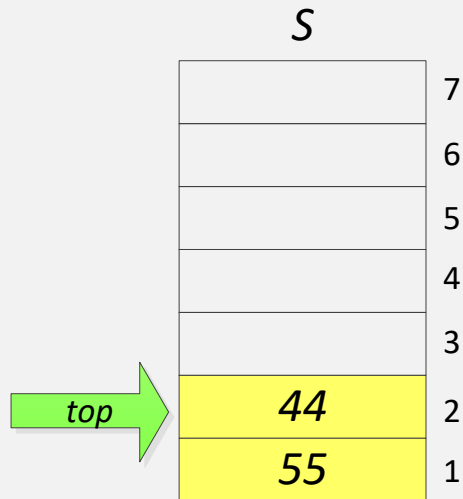
TIPOS DE DADOS ABSTRATOS (ADT)

- Definição de um **ADT**:
 - É caracterizado pelas operações que podem ser realizadas
 - Não indica como é que as operações estão implementadas
 - Não indica como é que os valores vão estar organizados na memória
 - Ou que algoritmos é que vão ser utilizados para implementar essas operações
 - Existem vários ADTs para descrever diferentes tipos de coleções de elementos, como por exemplo listas e conjuntos

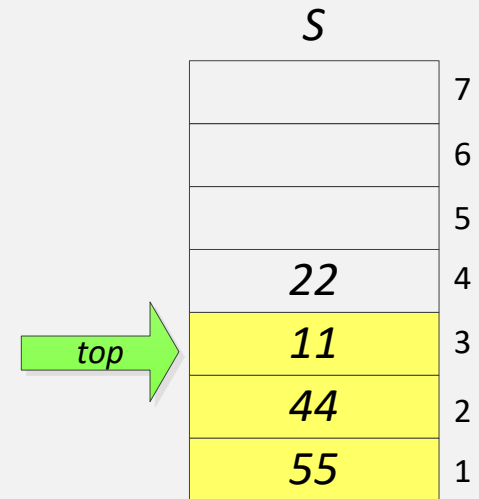
EXEMPLO DE UM ADT: STACK

- O **stack** (pilha) é uma estrutura de dados dinâmica cuja remoção de elementos está pré-definida
 - O próximo elemento a remover do conjunto é o que foi inserido mais recentemente
- 
- O *stack* implementa uma organização LIFO (*last-in-first-out*), com as seguintes operações:
 - **IsEmpty** – verifica se a pilha está vazia
 - **Push** – coloca um novo elemento no topo da pilha
 - **Pop** – retira o elemento que está no topo da pilha

STACK



Push(S, 11)
Push (S, 22)



x = Pop(S)
(devolve 22)

```
IsEmpty(S)
  if S.top == 0
    return TRUE
  else
    return FALSE
```

```
Push (S, x)
  S.top = S.top + 1
  S[S.top] = x
```

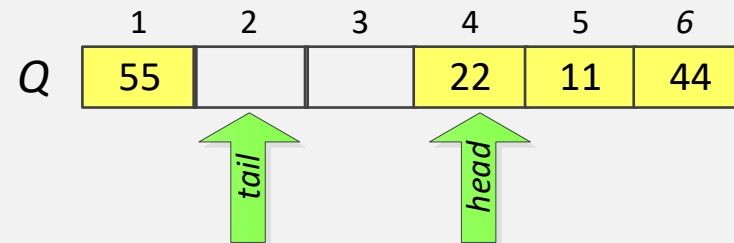
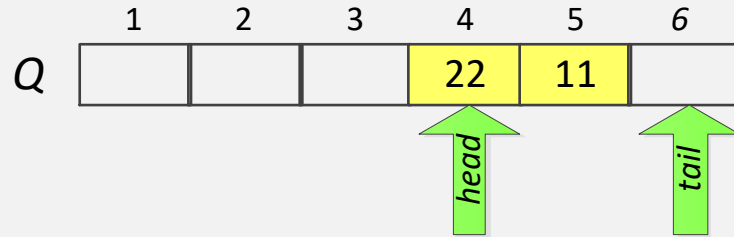
```
Pop (S)
  if IsEmpty(S)
    error "underflow"
  else
    S.top = S.top - 1
    return S[S.top + 1]
```

EXEMPLO DE UM ADT: *CIRCULAR QUEUE*

- A *circular queue* (fila circular) é uma estrutura de dados dinâmica cuja remoção de elementos está pré-definida
 - O próximo elemento a remover é sempre o que está à mais tempo no conjunto
-
- A *circular queue* implementa uma organização FIFO (*first-in-first-out*), com as seguintes operações:
 - **Enqueue** – insere um novo elemento na fila
 - **Dequeue** – remove o primeiro elemento da fila



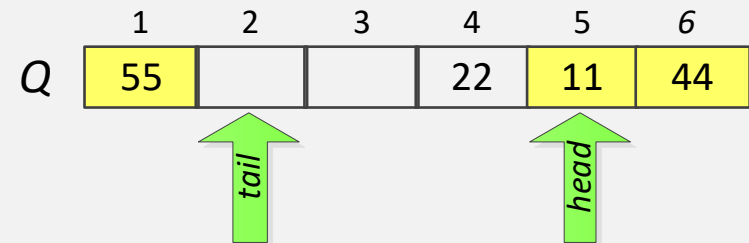
CIRCULAR QUEUE



Enqueue(Q, 44)
Enqueue (Q, 55)

```

Enqueue(Q, x)
  Q[Q.tail] = x
  if Q.tail == Q.length
    Q.tail = 1
  else
    Q.tail = Q.tail + 1
  
```



x = Dequeue(Q)
(devolve 22)

```

Dequeue(Q)
  x = Q[Q.head]
  if Q.head == Q.length
    Q.head = 1
  else
    Q.head = Q.head + 1
  return x
  
```

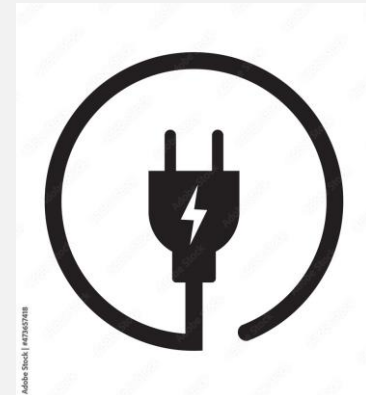
TIPOS DE DADOS ABSTRATOS (ADT)

- Definição de **API (Application Programming Interface)**
 - A **API** de uma **ADT**
 - Define uma lista de operações com uma descrição do objetivo de cada uma das operações
 - É um **contrato** que deve ser seguido por todas as implementações do ADT
 - As **API's** para descreverem as coleções de elementos em *kotlin.collections* e em *java.util*
 - São definidas através de **interfaces**
 - E as implementações são concretizadas através de **classes**

INTERFACES E CLASSES

- Através da **interface**
 - Define-se um **contrato** para um tipo de dados
 - Ou seja, define-se um conjunto de propriedades e comportamentos que os tipos de dados concretos devem implementar
- Exemplo: interface **Pluggable**

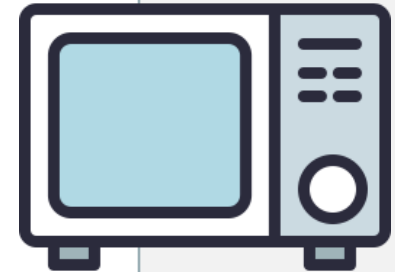
```
interface Pluggable {  
    val neededWattToWork: Int  
  
    //Measured in Watt  
    fun electricityConsumed(wattLimit: Int) : Int  
    fun turnOff()  
    fun turnOn()  
}
```



INTERFACES E CLASSES

- Os tipos de dados concretos são definidos através de **classes**
 - Os dados, tomam a forma de **propriedades** e o comportamento é implementado por funções designadas por **métodos**
 - Exemplo: a classe **Microwave** implementa a interface **Pluggable**:

```
class Microwave : Pluggable {  
    override val neededWattToWork = 15  
  
    override fun electricityConsumed(wattLimit: Int): Int {  
        return if (neededWattToWork > wattLimit) turnOff()  
        else {  
            turnOn()  
            neededWattToWork  
        }  
    }  
    override fun turnOff() {  
        println("Turning off..")  
    }  
  
    override fun turnOn() {  
        println("Turning on..")  
    }  
}
```



INTERFACES E CLASSES

- Os tipos de dados concretos são definidos através de **classes**
 - Os dados, tomam a forma de **propriedades** e o comportamento é implementado por funções designadas por **métodos**
 - Exemplo: a classe **Microwave** implementa a interface **Pluggable**:

```
class WashingMachine : Pluggable {  
    override val neededWattToWork = 60  
  
    override fun electricityConsumed(wattLimit: Int): Int {  
        return if (neededWattToWork > wattLimit) turnOff()  
        else {  
            turnOn()  
            neededWattToWork  
        }  
    }  
    override fun turnOff() {  
        println("Turning off..")  
    }  
  
    override fun turnOn() {  
        println("Turning on..")  
    }  
}
```



GENÉRICOS

- O tipo **genérico** é um tipo que pode ser parametrizado por outros tipos concretos
- Assim, podemos generalizar vários comportamentos do tipo de dados sem termos de nos comprometer com o tipo em concreto
- Utiliza-se normalmente um carater: **E**, **T**, **K**, **V**, ...

```
interface AEDStack<E> {  
    fun isEmpty(): Boolean  
    fun push(element: E)  
    fun pop(): E?  
}
```

GENÉRICOS

- A implementação das funções definidas na interface **AEDStack** é feita pela classe **AEDStackArray** considerando também os tipos de dados genéricos

```
class AEDStackArray<E>(capacity): AEDStack<E> {  
    private val elements = arrayOfNulls<Any?>(capacity) as Array<E?>  
    private var top = 0  
  
    override fun isEmpty() = top == 0  
  
    override fun push(element: E) {  
        if (top != elements.size)  
            elements[top++] = element  
    }  
    override fun pop() =  
        if (isEmpty()) null  
        else {  
            val element = elements[top - 1]  
            elements[--top] = null  
            element  
        }  
}
```

GENÉRICOS

- O tipo do elemento tem que ser conhecido em tempo de compilação para ser criada a instância de *array*. Deste modo, não é possível fazer:

XXXelements = arrayOfNulls<E?>(capacity)**XXX**

- Em Kotlin, qualquer tipo, seja Int ou String é considerado como sendo do tipo **Any**. Deste modo, em vez da solução anterior fica:

elements = arrayOfNulls<Any?>(capacity) as Array<E?>

- É requerido explicitamente a criação de um array do tipo Any e depois realiza-se um **type-cast** para **E?**

- Outro exemplo de **type-cast**:

```
var str1: Any = "This is a safe casting"
```

```
val str2: String? = str1 as? String
```

INTERFACES E CLASSES

- Exemplo: interface *AEDList*

```
interface AEDList<E> {  
    val size: Int  
    fun isEmpty(): Boolean  
    fun contains(element: E): Boolean  
    fun get(idx: Int): E?  
    fun set(idx: Int, value: E)  
}
```

size

INTERFACES E CLASSES

- Exemplo: classe *AEDArrayList* implementa a interface *AEDList*:

```
class AEDArrayList<E>(capacity: Int): AEDList<E> {  
    private val elements = arrayOfNulls<Any?>(capacity) as Array<E>
```

Construtor primário – a palavra *constructor* está implícita

```
    override val size: Int
```

```
        get() = elements.size
```

```
    override fun isEmpty() = elements.isEmpty()
```

```
    override fun contains(element: E) = elements.contains(element)
```

```
    override fun get(idx: Int) =
```

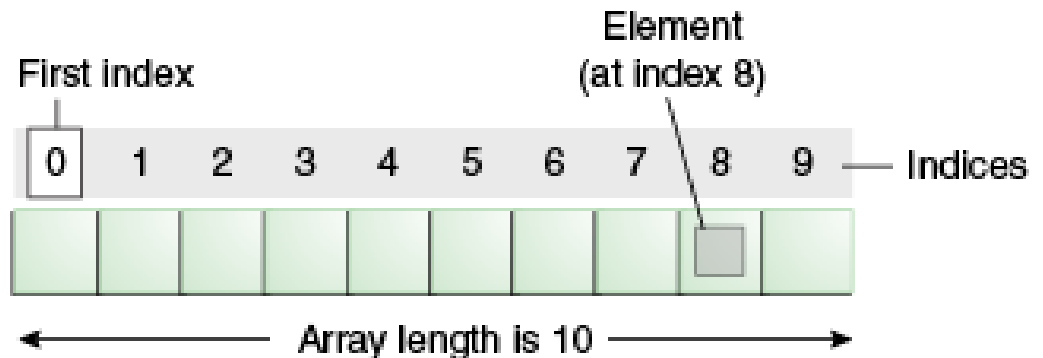
```
        if (idx in elements.indices) elements[idx] else null
```

```
    override fun set(idx: Int, value: E) {
```

```
        if (idx in elements.indices) elements[idx] = value
```

```
    }
```

```
}
```



INTERFACES E CLASSES

- Para definir construtores secundários usa-se a palavra chave: *constructor*

```
class AEDArrayList<E>: AEDList<E> {  
    private val elements: Array<E?>
```

Construtor secundário

```
    constructor() {  
        elements = arrayOfNulls<Any?>(10) as Array<E?>  
    }
```

```
    constructor(capacity: Int) {  
        elements = arrayOfNulls<Any?>(capacity) as Array<E?>  
    }
```

Construtor secundário

```
    override val size: Int  
        get() = elements.size
```

```
    override fun isEmpty() = elements.isEmpty()
```

```
    override fun contains(element: E) = elements.contains(element)
```

```
    override fun get(idx: Int) =  
        if (idx in elements.indices) elements[idx] else null
```

```
    override fun set(idx: Int, value: E) {  
        if (idx in elements.indices) elements[idx] = value  
    }
```

```
}
```

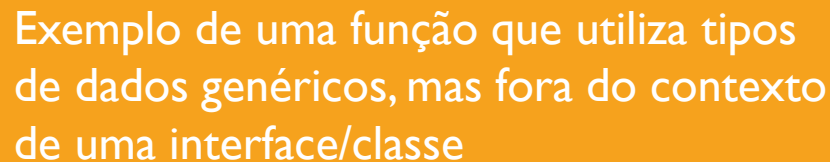
INTERFACE COMPARATOR

- Interface *Comparator<E>*
 - Providencia a função de comparação `compare()` que define a ordenação entre elementos do tipo genérico
 - `fun compare(a: E, b: E): Int`
 - Compara os dois argumentos a e b, devolvendo
 - zero se os argumentos forem iguais
 - um número positivo se $a > b$
 - um número negativo se $a < b$

INTERFACE COMPARATOR

- Exemplo de utilização da interface *Comparator*

```
fun main() {  
  
    val cmpNumbers = { n1: Int, n2: Int -> n1 - n2 }  
    val x = 9  
    val y = 11  
    println(compareAny(x, y, cmpNumbers))  
  
    val cmpStrings = { s1: String, s2: String -> s1.compareTo(s2) }  
    val name1 = "Ana"  
    val name2 = "Tiago"  
    println(compareAny(name1, name2, cmpStrings))  
  
}
```



Exemplo de uma função que utiliza tipos de dados genéricos, mas fora do contexto de uma interface/classe

```
fun <E> compareAny(value1: E, value2: E, cmp: Comparator<E>) =  
    cmp.compare(value1, value2)
```

INTERFACE COMPARATOR

- Exemplo utilizando um tipo de dados estruturado: **Pair<I,V>**

```
fun main() {  
  
    val cmpDices = { d1: Pair<Int, String>, d2: Pair<Int, String> -> d1.first - d2.first }  
    val dice1 = Pair(3,"um")  
    val dice2 = Pair(5,"cinco")  
    println(compareAny(dice1, dice2, cmpDices))  
}  
  
fun <E> compareAny(value1: E, value2: E, cmp: Comparator<E>) =  
    cmp.compare(value1, value2)
```

HERANÇA

- Quando se pretende definir uma interface (*AEDMutableList*) que contenha as características de outra interface (*AEDList*), mas que acrescenta mais características específicas a um determinado contexto (ex: contexto da mutabilidade)
- então define-se que a interface *AEDMutableList* herda de *AEDList*

```
interface AEDList<E> {  
    val size: Int  
    fun isEmpty(): Boolean  
    fun contains(element: E): Boolean  
    fun get(idx: Int): E?  
    fun set(idx: Int, value: E)  
}
```

```
interface AEDMutableList<E>: AEDList<E> {  
    fun add(element: E): Boolean // adiciona no fim da lista  
    fun remove(element: E, cmp: Comparator<E>): Boolean  
}
```

HERANÇA

- Classe *AEDMutableArrayList*

```
class AEDMutableArrayList<E>: AEDMutableList<E> {  
    private var elements: Array<E?>  
  
    constructor() {  
        elements = arrayOfNulls<Any?>(10) as Array<E?>  
    }  
    constructor(capacity: Int) {  
        elements = arrayOfNulls<Any?>(capacity) as Array<E?>  
    }  
    override var size: Int = 0  
  
    override fun isEmpty() = size == 0  
  
    override fun contains(element: E) = elements.contains(element)  
  
    override fun get(idx: Int) =  
        if (idx in elements.indices) elements[idx] else null  
  
    override fun set(idx: Int, value: E) {  
        if (idx in elements.indices) elements[idx] = value  
    }  
}
```

HERANÇA

- Classe *AEDMutableArrayList*

```
override fun add(element: E): Boolean {  
    if (size == elements.size)  
        increaseCapacity()  
    elements[size++] = element  
    return true  
}  
  
private fun increaseCapacity() {  
    val newArray = arrayOfNulls<Any?>(2 * elements.size) as Array<E?>  
    // arraycopy(src, srcStart, dst, dstStart, size)  
    System.arraycopy(elements, 0, newArray, 0, size)  
    elements = newArray  
}
```

HERANÇA

- Classe *AEDMutableArrayList*

```
override fun remove(element: E, cmp: Comparator<E>): Boolean {  
    for (i in elements.indices)  
        if (cmp.compare(element, elements[i]) == 0) {  
            size--  
            System.arraycopy(elements, i+1, elements, i, size - i)  
            return true  
        }  
    return false  
}
```