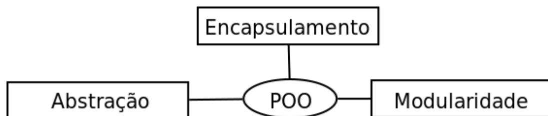
Tipos Abstractos de Dados

Pedro Ribeiro

DCC/FCUP

2019/2020



(baseado e/ou inspirado parcialmente nos slides de Luís Lopes e de Fernando Silva)

Programação Orientada a Objectos

- **Programação orientada a objectos** é um paradigma onde os principais actores são os **objectos**
- Três dos seus principais **princípios** são:
  - **Abstração** - Especificação do que um objecto deve fazer, mas não como o fazer (tipos abstractos de dados) (**o quê** *vs* **como**)
  - **Encapsulamento** - Esconder os detalhes internos de cada componente de um programa *(código independente de outros objectos)*
  - **Modularidade** - Divisão de um sistema em diferentes componentes funcionais *(mais robustez e facilidade de debug)*
- Estes três principio ajudam a melhorar **robustez**, **adapatibilidade** e aumentam o potencial de **reutilização** do código.

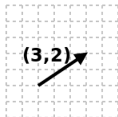
Tipos Abstractos de Dados (TADs)

- um **Tipo Abstracto de Dados** (**TAD**) é "modelo" de um tipo de dados definido pelo seu comportamento esperado:
  - Quais os **valores** que pode conter
  - Quais as **operações** permitidas sobre esses valores
- um TAD apenas define a **API** (*Application Programming Interface*)
  - Quais os **métodos** para interagir com o objecto
- diz-se **abstracto** porque a sua definição não carece de uma implementação concreta
  - para um mesmo TAD podem existir várias implementações possíveis
  - ex.: uma pilha pode ser implementada com arrays ou com listas, um conjunto pode ser implementar com arrays ou com árvores, (vamos ver vários exemplos durante o semestre)
- O Java suporta TADs através do conceito de **interfaces** e de **classes**

Vectores

- Vejamos exemplo inicial de um Tipo Abstracto de Dados (TAD)
- Imagine que quer representar um vector em $\mathbb{R}^2$:

  Por exemplo:

- Precisamos de saber:
  - Quais os dados que armazena? *(atributos da classe)*
  - Como podemos interagir com objecto? *(métodos da classe)*

Vectores - Atributos

- Um vector fica definido por um par $(\mathbf{x}, \mathbf{y})$, que podemos representar por dois atributos
- Vamos usar **doubles** e não *ints*, porque são vectores em $\mathbb{R}^2$

```
class Vector {
   // Atributos (coordenadas x e y)
   double x, y;

   // Construtor
   Vector(double x0, double y0) {
      x = x0;
      y = y0;
   }
}

class TestVector {
   public static void main(String[] args) {
      Vector v1 = new Vector(3,2); // Exemplo de criação de vectores
      Vector v2 = new Vector(1,4);
   }
}
```

Vectores - Sobre a palavra chave "this"

- Notem como se usaram nos construtores argumentos com nomes diferentes dos atributos (para não haver ambiguidade)

```
class Vector {
   double x, y;
   // x0 tem nome diferente de x, y0 tem nome diferente de y
   Vector(double x0, double y0) {
      x = x0;
      y = y0;
   }
}
```

- Se precisarmos de desambiguar, ou de nos referirmos ao objecto si, à "nossa própria instância", podemos usar a palavra chave **this**:

```
class Vector {
   double x, y;
   // o atributo do objecto x fica igual ao argumento do construtor x
   // (a mesma coisa com o y)
   Vector(double x, double y) {
      this.x = x;
      this.y = y;
   }
}
```

Vectores - Sobre a palavra chave "private"

- A declaração dos atributos não impede outra classe de aceder a eles

```
class Vector {
   double x, y;

   Vector(double x0, double y0) {
      x = x0;
      y = y0;
   }
}

class TestVector {
   public static void main(String[] args) {
      Vector v1 = new Vector(3,2);
      v1.x = 4; // Esta linha funciona e muda o valor de x
   }
}
```

- No que concerne ao **encapsulamento** dos dados, isto não é desejável:,
  - Outras classes podem aceder/modificar o estado interno da classe
  - Se quisermos mudar a implementação, podemos quebrar outro código
  - Queremos promover interação com a classe apenas com os métodos!

Vectores - Sobre a palavra chave "private"

- Podemos declarar atributos como sendo privados usando a palavra-chave **private** para classificar as variáveis

```
class Vector {
   private double x, y; // os atributos são agora
                        // classificados como "privados"
   Vector(double x0, double y0) {x = x0; y = y0;}
}

class TestVector {
   public static void main(String[] args) {
      Vector v1 = new Vector(3,2);
      v1.x = 4; // Erro de compilação:  "x has private access in Vector"
}
```

- Para qualificar uma variável ou método podemos usar:
(quando e se necessário iremos explicar os outros qualificadores)

Accessível em:	**public**	**protected**	**default**	**private**
mesma classe	Sim	Sim	Sim	Sim
classe do mesmo package	Sim	Sim	Sim	Não
sub-classe num package diferente	Sim	Sim	Não	Não
não sub-classe em package diferente	Sim	Não	Não	Não

Vectores - Sobre a palavra chave "private"

- Para que uma outra classe possa agora aceder aos dados, temos de criar métodos. Os métodos para aceder aos atributos são os "*getters*":

```
class Vector {
   // Atributos
   private double x, y;

   // Construtor
   Vector(double x0, double y0) {x = x0; y = y0;}

   // Getters
   public double getX() {return x;}
   public double getY() {return y;}
}

class TestVector {
   public static void main(String[] args) {
      Vector v1 = new Vector(3,2);
      double valor = v1.getX(); // Acedendo ao valor de X
   }
}
```

- Notem que só podemos "ler" o valor dos atributos e não modificar (se quisessemos mudar o valor criávamos métodos "*setters*")

Vectores - Como escrever o conteúdo de uma classe

- Como sabem, se imprimirmos uma variável de classe, iremos ver apenas a referência e não o conteúdo:

```
class Vector {
   // Atributos
   private double x, y;

   // Construtor
   Vector(double x0, double y0) {x = x0; y = y0;}
}

class TestVector {
   public static void main(String[] args) {
      Vector v1 = new Vector(3,2);
      System.out.println(v1);
   }
}
```

- O output do programa anterior seria algo como: `Vector@75b84c92`
- É possível dar à classe a capacidade de ser "imprimida" de maneira mais fácil do que ter de usar os *getters* para ir buscar o conteúdo.

Vectores - Como escrever o conteúdo de uma classe

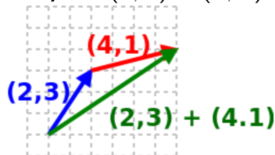
- Se adicionarmos uma classe "`public String toString()`" à nossa classe, mudamos o comportamento quando é esperada uma String
  (como acontece quando a variável é chamada numa instrução de impressão)

```
class Vector {
   // (...)

   //Conversão de um vector para String
   public String toString() {
      return "(" + x + "," + y + ")";
   }
}

class TestVector {
   public static void main(String[] args) {
      Vector v1 = new Vector(3,2);
      System.out.println(v1);
   }
}
```

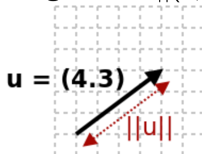
- O output do programa é agora: `(3.0,2.0)`
- Tecnicamente estamos a fazer **override** do método padrão de impressão de um objecto (que é simplesmente `NomeClasse@HashCode`)

Vectores - Algumas operações

- Vamos agora adicionar alguma funcionalidade aos nossos vectores, adicionando alguns operadores:
  - **Adição:** $(a, b) + (c, d) = (a + c, b + d)$



  - **Subtração:** $(a, b) - (c, d) = (a - c, b - d)$
  - **Magnitude:** $||(a, b)|| = \sqrt{a^2, b^2}$



  - **Multiplicação por um escalar:** $(a, b) \times c = (a \times c, b \times c)$

Vectores - Algumas operações

- Precisamos de decidir como vamos usar os operadores:
  - Queremos que alterem o vector em si
    vs
  - Queremos que o resultado seja um novo vector
- Vamos usar aqui a segunda hipótese
- Os métodos de uma classe têm acesso aos seus atributos
  (não é preciso usar *getters*)

```
class Vector {
   // (...)

   public Vector add(Vector v) { return new Vector(x + v.x, y + v.y); }

   public Vector sub(Vector v) { return new Vector(x - v.x, y - v.y); }

   public double magnitude() { return Math.sqrt(x*x + y*y); }

   public Vector scale(double c) { return new Vector(x*c,y*c); }
}
```

Vectores - Exemplo de utilização

```
class TestVector {
   public static void main(String[] args) {
      Vector v1 = new Vector(3,2);
      Vector v2 = new Vector(1,4);

      Vector v3 = v1.add(v2);
      Vector v4 = v1.sub(v2);
      double  m = v1.magnitude();
      Vector v5 = v1.scale(2);

      System.out.println(v3);
      System.out.println(v4);
      System.out.printf("%.2f%n", m);
      System.out.println(v5);
   }
}
```

Output:

```
(4.0,6.0)
(2.0,-2.0)
3.61
(6.0,4.0)
```

TAD Conjunto

Vamos agora definir um novo TAD: um **conjunto**

- Representa um conjunto de elementos (**sem repetições**)
- Suporta operações como **adicionar** um elemento, **remover** um elemento, **verificar** se um elemento está no conjunto, etc

Este TAD é muito **útil** como *peça base* em vários algoritmos. Exemplos:

- Quantos elementos diferentes existem no *input*?
  É só colocar todos num conjunto e ver o seu tamanho.
- Quero fazer uma pesquisa por um conjunto de *sítios*, mas não quero voltar a um *sítio* onde já estive?
  Guardo os sítios visitados num conjunto e uso a operação de verificar.

TAD Conjunto - Operações

Vamos agora implementar este TAD.

Queremos implementar as seguintes operações básicas:
(existiriam mais operações possíveis, como a união, interseção, etc)

- **boolean contains(x)** - verifica se o elemento x está no conjunto. Retorna *true* se o elemento está no conjunto e *false* caso contrário.
- **boolean add(x)** - adiciona o elemento x ao conjunto. Retorna *true* se foi adicionado ou *false* caso x já esteja no conjunto.
- **boolean remove(x)** - remove o elemento x do conjunto. Retorna *true* se foi removido ou *false* caso x não esteja no conjunto.
- **int size()** - retorna o número de elementos do conjunto.
- **void clear()** - limpa o conjunto (torna-o vazio)

TAD Conjunto - Exemplo de utilização

Um exemplo de utilização e do significado das operações.
Seja S um conjunto.

- Inicialmente $S = \emptyset$
- $S.add(1)$ iria devolver *true* e tornar $S = \{1\}$
- $S.add(5)$ iria devolver *true* e tornar $S = \{1, 5\}$
- $S.add(7)$ iria devolver *true* e tornar $S = \{1, 5, 7\}$
- $S.contains(1)$ iria devolver *true*
- $S.contains(2)$ iria devolver *false*
- $S.add(1)$ iria devolver *false* e continuaríamos com $S = \{1, 5, 7\}$
- Nesta fase $S.size()$ iria devolver 3, o número de elementos de S
- $S.remove(5)$ iria devolver *true* e tornar $S = \{1, 7\}$
- $S.remove(5)$ iria devolver *false* e continuaríamos com $S = \{1, 7\}$
- $S.clear()$ iria tornar $S = \emptyset$

TAD Conjunto - Interface

Por uma questão de simplificação, iremos implementar um **conjunto de números inteiros**.
(mais para a frente vamos falar de genéricos e como poderíamos implementar um conjunto de "qualquer tipo")

O Java permite usar a noção de **interface** para especificar uma API (a assinatura dos métodos) sem explicar como implementar:

- Isto permite uma verdadeira **abstração** do TAD
- Permite que uma classe tenha um objecto do tipo desse interface e saiba que tem acesso a toda a API
- Obriga uma classe que implemente esse interface a declarar todos os métodos definidos na API

TAD Conjunto - Interface

Um interface em Java essencialmente é uma declaração dos métodos sem especificar o seu *corpo* (o código que implementa o método).

```
interface IntSet {
   public boolean contains(int x); // Retorna true se x está no conjunto
   public boolean add(int x);      // Adiciona x ao conjunto
   public boolean remove(int x);   // Remove x do conjunto
   public int     size();          // Retorna o nr de elementos do conjunto
   public void    clear();         // Limpa o conjunto (torna-o vazio)
}
```

Dentro de um outro programa, podemos ter variáveis do tipo do interface;

```
IntSet s;
```

ou num outro exemplo possível de utilização:

```
void doSomething(IntSet s)
```

TAD Conjunto - Interface

Um interface não tem construtores:

- **Não permite criar (instanciar) um novo objecto desse tipo**
- Para isso precisamos de ter a implementação em si

Uma classe pode dizer que **implementa um interface**:

```
class Xpto implements IntSet {
  // (...)
}
```

Se a classe não implementar todos os métodos do Interface o Java gera um erro de compilação: `Xpto is not abstract and does not override abstract method clear() in IntSet`

Tendo a classe definida podemos instanciar objectos desse tipo:

```
IntSet s = new Xpto();
```

Como *s* é do tipo *IntSet* apenas poderá chamar métodos do interface (mesmo que Xpto implemente outros métodos)

TAD Conjunto - Implementando um Interface

Vamos agora implementar então o interface *IntSet* como atrás definido.

Primeira coisa a pensar é **como** implementar:

- Existem muitas hipóteses de implementação com diferentes vantagens e desvantagens
  - Ex: qual o tempo de execução? quais os gastos de memória?
- Não existe implementação "perfeita": dependendo do uso, uma pode ser melhor que outra
- Nesta UC vamos precisamente querer perceber as implementações para perceber os vários *tradeoffs* e saber escolher a melhor para o nosso caso em particular
- Vamos começar por usar um array

Que ideias têm sobre como implementar um IntSet?

TAD Conjunto - Uma lista num array

Comecemos por usar estrutura de dados que conhecemos bem: um **array**!

- Vamos manter num array *elem* uma lista dos elementos
- O tamanho do array determina o número máximo de elementos
- Mantemos numa outra variável *size* o número de elementos

Por exemplo, o conjunto $\{1, 5, 7\}$ seria representado por:

elem =

1	5	7							

$size = 3$

Operacionalizando isto, os **atributos** seriam:

```
// Implementa um conjunto usando um array como lista de elementos
class ArrayListIntSet implements IntSet {
   private int size;   // Numero de elementos do conjunto
   private int elem[]; // Array que contem os elementos em si

   // (...)
}
```

TAD Conjunto - Uma lista num array

(os métodos que se seguem são para serem colocados dentro da classe ArrayListIntSet)

- Como poderia ser o **construtor**?

```
// Construtor que recebe como argumento o número máximo de elementos
ArrayListIntSet(int maxSize) {
   elem = new int[maxSize];
   size = 0;
}
```

- Devolver o **nº de elementos** é só ir buscar o valor da variável *size*:

```
public int size() {
   return size;
}
```

- **Limpar o conjunto** é só colocar a variável *size* em zero

  (não é preciso limpar *elem*[] porque *size* determina as posições que "interessam")

```
public void clear() {
   size = 0;
}
```

TAD Conjunto - Uma lista num array

- Como **verificar** se um elemento está no conjunto?

```
public boolean contains(int x) {
   for (int i=0; i<size; i++)
      if (elem[i] == x)
         return true;
   return false;
}
```

- Como **adicionar** um elemento?
  (podemos aproveitar outros métodos, como o *contains*())

```
public boolean add(int x) {
   if (!contains(x)) {
      elem[size] =  x;
      size++;
      return true;
   }
   return false;
}
```

TAD Conjunto - Uma lista num array

- Como **remover** um elemento?
  - Não basta retirar o elemento (o que fica na sua anterior posição?)
  - Podemos colocar o último elemento nessa posição que ficou vazia! *(a ordem dos elementos não interessa num conjunto)*
  - Feito isso, resta reduzir o *size*

```
public boolean remove(int x) {
   if (contains(x)) {
      int pos = 0;
      while (elem[pos] != x) pos++; // descobrir posicao de x
      size--;
      elem[pos] = elem[size]; // Trocar último elemento
      return true;            // com o que se removeu
   }
   return false;
}
```

Exemplo:

Seja $S = \{1, 3, 5, 7, 9\}$, $elem =$ | 1 | 5 | 3 | 9 | 7 | e $size = 5$

$s.remove(5)$ leva a $S = \{1, 3, 7, 9\}$, $elem =$ | 1 | 7 | 3 | 9 | | e $size = 4$

TAD Conjunto - Uma lista num array

- Já agora, como pode poderíamos **converter em String**?
  (para poder escrever por exemplo com System.out.println())

```
public String toString() {
   String res = "{";
   for (int i=0; i<size; i++) {
      if (i>0) res += ",";
      res += elem[i];
   }
   res += "}";
   return res;
}
```

TAD Conjunto - Exemplo de utilização

Com tudo isto, um exemplo de utilização seria:

```
public class TestSet {
   public static void main(String[] args) {
      IntSet s = new ArrayListIntSet(100);

      System.out.println(s);
      System.out.println(s.add(1));
      System.out.println(s.add(5));
      System.out.println(s.add(7));
      System.out.println(s);
      System.out.println(s.contains(1));
      System.out.println(s.contains(2));
      System.out.println(s.add(1));
      System.out.println(s.size());
      System.out.println(s.remove(5));
      System.out.println(s.remove(5));
      System.out.println(s);
      s.clear();
      System.out.println(s);
      System.out.println(s.size());
   }
}
```

TAD Conjunto - Vantagens e Desvantagens

Sobre a implementação:

- Algumas **vantagens**
  - **Simples** de implementar (não menosprezar este factor!) *(implementações mais complicadas mais facilmente contêm erros)*
  - Não gasta muita **memória** *(se soubermos o número de elementos máximo, temos só o espaço necessário para esse maximo)*
- Algumas **desvantagens**
  - Operações de adicionar, remover e verificar implicam **percorrer array** à procura do elemento *(tempo depende do número de elementos no conjunto)*
  - Tem um **limite máximo** no número de elementos *(poderíamos criar novo array quando ficarmos sem espaço, mas implica copiar elementos)*

TAD Conjunto - Sobre os erros

Como **nota adicional**, poderíamos criar um novo tipo de erro (excepção) do nosso próprio conjunto. Considerem por exemplo o seguinte código:

```
public class TestSet {
   public static void main(String[] args) {
      IntSet s = new ArrayListIntSet(1);
      s.add(1);
      s.add(2);
   }
}
```

Isto vai gerar um erro de execução por acedermos fora dos limites do array:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 1
       at ArrayListIntSet.add(TestSet.java:22)
       at TestSet.main(TestSet.java:127)
```

Se alguém estiver a usar o nosso TAD isto não é muito informativo e depende da nossa implementação...

TAD Conjunto - Sobre os erros

Podemos gerar um erro de execução com um erro "customizado":

```
throw new RuntimeException("Mensagem de erro customizada")
```

Por exemplo, poderiamos alterar o *add* para:

```
public boolean add(int x) {
   if (!contains(x)) {
      if (size == elem.length)
         throw new RuntimeException("Maximum size of set reached");
      elem[size] =  x;
      size++;
      return true;
   }
   return false;
}
```

Agora o código anterior já daria o seguinte erro:

```
java.lang.RuntimeException: Maximum size of set reached
```

Voltaremos a falar de erros e do mecanismo de excepções noutras aulas

TAD Conjunto - Array de booleanos

Vamos agora abordar outra implementação para o TAD conjunto, que é **mais rápida** a adicionar, remover e verificar.

Uma ideia é usar um **array de booleanos** para dizer se um número está ou não no conjunto.

- Vamos manter um array *isElem* de valores booleanos
- $isElem[i]$ diz-nos se i está ou não no conjunto
- O tamanho do array determina o tamanho do número máximo
- Mantemos numa outra variável *size* o número de elementos

Por exemplo, o conjunto $\{1, 5, 7\}$ seria representado por:

$isElem =$

0	1	2	3	4	5	6	7
F	T	F	F	F	T	F	T

$size = 3$

TAD Conjunto - Array de booleanos

A implementação desta "versão" do TAD conjunto é um dos objectivos das **aulas práticas** desta semana.

Em comparação com a implementação anterior de lista como array:

- Algumas **vantagens**
  - É muito mais **rápida** a adicionar, remover e verificar *(tempo constante, não depende do número de elementos do conjunto)*
- Algumas **desvantagens**
  - Gasta **mais memória**
    - Precisa de guardar explicitamente os elementos que não estão
    - Limita o tamanho (magnitude) dos números a guardar
  - **Menos "generalizável"**:
    - Usa os números como índices
    - Se o que guardarmos não fossem números, como saber a sua posição no array?

TAD Conjunto

Na prática, para guardar conjuntos existem ainda outras soluções (mais eficientes) que não vamos detalhar agora, como por exemplo:

- Árvores Binárias Equilibradas (ex: AVL ou Red-Black)
- Tabelas de Dispersão (a.k.a. tabelas de *hash*)

O Java providencia implementações do TAD Conjunto prontas a usar:
https://docs.oracle.com/javase/tutorial/collections/implementations/set.html

- TreeSet (usa árvores)
- HashSet (usa *hash tables*)
- LinkedHashSet (usa *hash tables* e *listas ligadas* para percorrer)

No contexto de **Estruturas de Dados**, vamos querer implementar os nossos TADS (e não "simplesmente" usar os do Java) precisamente porque esse é o objectivo desta UC. Saber como implementar permite:

- Perceber realmente os *tradeoffs* que implicam uma escolha
- Customizar/aumentar a estrutura de dados às nossas necessidades
- Conseguir criar novas estruturas de dados consoante o necessário

TAD Conjunto - TreeSet de Java

Um exemplo de uso dos conjuntos de Java:

- **Set:** um interface
- **TreeSet:** uma implementação do interface Set

```
import java.util.Set;
import java.util.TreeSet;

public class TestTreeSet {
   public static void main(String[] args) {
      Set<Integer> s = new TreeSet<Integer>();
      s.add(1);
      s.add(5);
      s.add(7);
      System.out.println(s.size());       // 3
      System.out.println(s.contains(1)); // true
      System.out.println(s.contains(2)); // false
      s.remove(5);
      System.out.println(s.size());       // 2
   }
}
```

https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html

Genéricos

- Esta linha tem algo de que ainda não falamos:

```
Set<Integer> s = new TreeSet<Integer>();
```

- Muitas das estruturas de dados de Java usam esta sintaxe:

```
NomeTAD<tipo> variavel
```

- Estas linhas usam a noção de tipos **genéricos**
  - Por vezes precisamos de algo que funcione para qualquer tipo
  - A implementação fica "genérica"
  - Concretizamos o tipo na declaração da variável (como em cima)
  - Poderíamos por exemplo ter antes um conjunto de Strings:

```
Set<String> s = new TreeSet<String>();
```

Exemplo de implementação com Genéricos

- Vejamos um exemplo de uso de uma implementação com genéricos
- Suponhamos que queremos ter um **"par" de objectos**, que funcione com qualquer tipo. Exemplos:
  - Um estudante com nome e número: (String,Integer)
  - Um ponto 2D: (Integer, Integer)
  - Um filme com nome e lista de actores: (String, TreeSet<String>)
  - Um vector com nome: (Vector, String)
  - ...
- Como podemos implementar uma classe **Pair** que funcione genericamente sem ser preciso reimplementar para cada combinação de tipos?

Um par de objectos usando genéricos

```
public class Pair<A,B> { // Uma implementação genérica de um par
   private A first;  // Objecto do tipo A
   private B second; // Objecto do tipo B

   Pair(A a, B b) {
      first = a;
      second = b;
   }

   public A getFirst()  { return first; }
   public B getSecond() { return second; }
}
```

```
public class TestPair { // Exemplo de uso da classe Pair
   public static void main(String[] args) {
      Pair<String, String>   p1 = new Pair<String,String>("UC","ED");
      Pair<Integer, Integer> p2 = new Pair<Integer,Integer>(42,1);
      Pair<String, Integer>  p3 = new Pair<String,Integer>("Dois",2);

      String s1 = p1.getFirst();  // "UC"
      String s2 = p1.getSecond(); // "ED"
   }
}
```

Wrappers

- Os tipos A e B do Pair esperam um objecto (e não tipos primitivos).
- Como colocar um tipo primitivo onde é esperado um objecto?
- O Java disponibiliza **Wrappers**: classes cujo objectivo é encapsular um tipo primitivo: **Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**, **Character**, **Boolean**.

```
public class TestWrappers {
   public static void main(String[] args) {
      Integer i = new Integer(42);
      Double  d = new Double(2.3);

      System.out.println(i.intValue());    // 42
      System.out.println(d.doubleValue()); // 2.3
   }
}
```

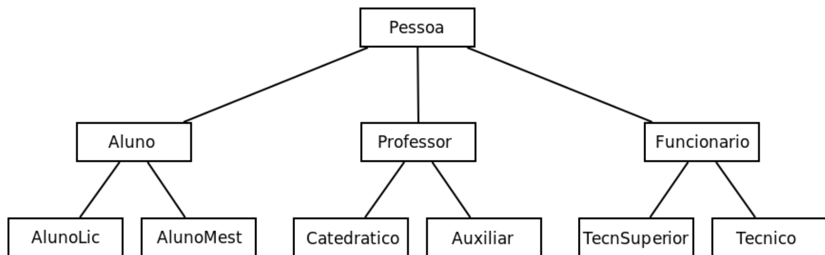
"Boxing" e "unboxing" automáticos

- Para facilitar o uso, o Java faz **boxing** e **unboxing** automáticos, um processo que faz conversão implícita entre tipos primitivos e os correspondentes Wrappers.
- Um exemplo:
  - Onde é esperado um `Integer` e é passado um **int**, o Java cria um `Integer` contendo o **int**
  - Onde é esperado um **int** e é passado um `Integer`, o Java vai buscar o `intValue()`

```
public class TestBoxing {
   public static void main(String[] args) {
      Integer a;
      int b;

      a = 42;     // Automatic boxing de 42
      b = a * 2; // Automatic unboxing do valor de a
   }
}
```

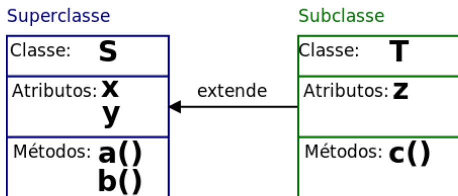
Herança

- Um outro mecanismo importante é o de **herança**
- Não vamos usar muito, mas é importante perceberem o **conceito**
- Uma forma natural de organizar componentes é numa **hierarquia** que vai desde o **mais geral** (no topo) até ao **mais específico** (em baixo)



Herança

- **Herança** é o mecanismo que permite implementar hierarquias, fazendo com que uma classe herde atributos e métodos de outra classe.
- Uma **subclasse** *extende* uma **superclasse** ou **classe base**.



A classe T ficaria com os atributos x, y e z, e os métodos $a()$, $b()$ e $c()$

Herança - Exemplo e uso de "extends" e "super"

```
class Person {
   private String name;
   private int age;

   Person(String n, int a) {name = n; age = a;}
   String getName() {return name;}
   int getAge() {return age;}
}
// Palavra chave "extends" indica herança
class Student extends Person {
   private int number;
   // "super" refere-se à super classe e chama o respectivo construtor
   Student(String n, int i, int m) {super(n,i); number = m;}
   int getNumber() {return number;}
}

class TestStudent {
   public static void main(String[] args) {
      Student a = new Student("Jose",20,12345);
      System.out.println(a.getName());   // Jose
      System.out.println(a.getAge());    // 20
      System.out.println(a.getNumber()); // 12345
   }
}
```

Herança - Variáveis e subclasses

- Uma variável pode referir-se a um objecto de uma subclasse do seu tipo (já o contrário não faz sentido).

```
class TestStudent {
   public static void main(String[] args) {
      // Note que agora estamos a dizer a alocar um Student a uma Person
      Person p = new Student("Jose",20,12345);
      System.out.println(p.getName());   // Jose
      System.out.println(p.getAge());    // 20

      // Esta linha daria um erro
      // System.out.println(p.getNumber());
   }
}
```

Herança - Override

- Uma subclasse pode "reimplementar" um método herdado (**override**)
- Para saber qual método executar, o Java percorre a hierarquia (de baixo para cima) até descobrir o método onde pode executar;

```
class TypeA {
   void write() {System.out.println("A");}
}

class TypeB extends TypeA {}

class TypeC extends TypeA {
   // É permitido: estamos a fazer override do método write
   void write() {System.out.println("C");}
}

class TestTypeABC {
   public static void main(String[] args) {
      TypeB b = new TypeB();
      TypeC c = new TypeC();
      b.write(); // Escreve "A"
      c.write(); // Escreve "C"
   }
}
```

Herança - Classes e Métodos abstractos

- Uma classe pode conter alguns métodos abstractos (não implementados) usando a palavra chave **abstract**
- Fica como um "misto" de interface com superclasse

```
abstract class TypeOne {
   void writeOne() {System.out.println("One");}
   abstract void writeTwo(); // Método abstracto (não implementado)
}
// Se não implentasse writeTwo() java queixava-se que tinha de ser "abstract"
class TypeTwo extends TypeOne {
   // Implementação do método abstracto de TypeA
   void writeTwo() {System.out.println("Two");}
}

class TestTypeOneTwo {
   public static void main(String[] args) {
      TypeTwo t = new TypeTwo();
      t.writeOne(); // Escreve "One"
      t.writeTwo(); // Escreve "Two"
   }
}
```

Heranças e Linguagem Java

- Agora já pode compreender melhor toda a hierarquia de classes de Java (onde a superclasse de topo é simplesmente `Object`):

https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html

Class TreeSet<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractSet<E> ⟵ Hierarquia de classes
            java.util.TreeSet<E>

Type Parameters:

E - the type of elements maintained by this set ⟵ Genéricos

All Implemented Interfaces: ⟵ Interfaces implementados

Serializable, Cloneable, Iterable<E>, Collection<E>, NavigableSet<E>, Set<E>, SortedSet<E>

```
public class TreeSet<E>
extends AbstractSet<E>
implements NavigableSet<E>, Cloneable, Serializable
```

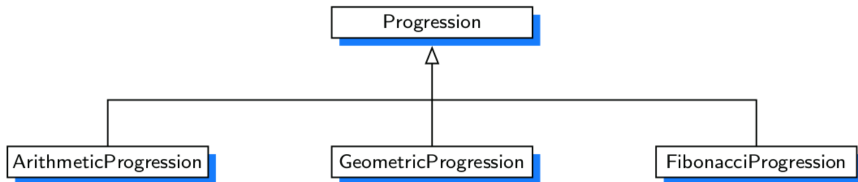
Exemplo de heranças: Progressões

Vamos agora ilustrar alguns dos conceitos dados:

- Vamos implementar **progressões** ou **sequências** de números inteiros
- O que define uma progressão é:
  - O número (ou números) inicial
  - A regra que define como obter cada próximo elemento da sequência
- Alguns exemplos:
  - **Progressão Aritmética:** $f(n) = f(n-1) + k$
    (cada termo é igual anterior mais um incremento fixo).
    - 0, 1, 2, 3, 4, 5, 6, 7, ... $(+1)$
    - 2, 7, 12, 17, 22, 27, ... $(+5)$
  - **Progressão Geométrica:** $f(n) = f(n-1) * k$
    (cada termo é igual anterior multiplicado por uma razão constante
    - 1, 2, 4, 8, 16, 32, 64, 128, ... $(\times 2)$
    - 2, 6, 18, 54, 162, 486, ... $(\times 3)$
  - **Fibonacci:** $f(n) = f(n-1) + f(n-2)$
    - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Exemplo de heranças: Progressões

- Vamos criar uma hierarquia de progressões



Queremos os seguintes métodos:

- **nextValue()** - Um método público para ir buscar o próximo valor da progressão, implicitamente avançando a progressão
- **printProgression(n)** - Um método público para imprimir *n* valores da progressão
- **advance()** - Um método protegido para avançar a progressão

Exemplo de heranças: Progressões

```
// Gera uma progressão.  Por omissão é simplesmente:  0, 1, 2, ...
public class Progression {
   protected long current; // valor actual
   // Por omissão começa em 0.  This chama construtor da própria classe
   Progression() { this(0); }
   // Constroi progressão començando num dado valor
   Progression(long start) { current = start; }
   // Devolve o valor actual e avança a sequência
   public long nextValue( ) {
      long answer = current;
      advance();
      return answer;
   }
   // Avança para o próximo valor da sequênca
   protected void advance( ) {
      current++;
   }
   // Escreve os próximos n valores da progressão
   public void printProgression(int n) {
      for (int i=0; i<n; i++)
         System.out.print(nextValue() + " ");
      System.out.println();
   }
}
```

Exemplo de heranças: Progressões

- Um exemplo de utilização da classe `Progression`

```
public class TestProgression {
   public static void main(String[] args) {
      Progression prog;

      System.out.print("Default Progression: ");
      prog = new Progression();
      prog.printProgression(10); // 0 1 2 3 4 5 6 7 8 9

      System.out.print("Progression with start 4: ");
      prog = new Progression(4);
      prog.printProgression(10); // 4 5 6 7 8 9 10 11 12 13
   }
}
```

Exemplo de heranças: Progressões Aritméticas

- Para além dos construtores, o único método que muda é... o advance()!

```java
public class ArithmeticProgression extends Progression {
   protected long increment;

   // Por omissão cria progressão: 0, 1, 2, ...
   public ArithmeticProgression() { this(1, 0); }

   // Cria progressão: 0, stepsize, 2*stepsize,...
   public ArithmeticProgression(long stepsize) { this(stepsize, 0); }

   // Cria progressão com um dado incremento e um dado início
   public ArithmeticProgression(long stepsize, long start) {
      super(start);
      increment = stepsize;
   }

   protected void advance( ) {
      current += increment;
   }
}
```

Exemplo de heranças: Progressões Geométricas

- Para além dos construtores, o único método que muda é... o advance()!

```
public class GeometricProgression extends Progression {
   protected long base;

   // Por omissão cria progressão: 1, 2, 4, 8, 16, ...
   public GeometricProgression() { this(2, 1); }

   // Cria progressão: 1, b, b^2, b^3, ...
   public GeometricProgression(long b) { this(b, 0); }

   // Cria progressão com uma dada base e um dado início
   public GeometricProgression(long b, long start) {
      super(start);
      base = b;
   }

   protected void advance( ) {
      current *= base;
   }
}
```

Exemplo de heranças: Sequência de Fibonacci

- Para além dos construtores, o único método que muda é... o advance()!

```
public class FibonacciProgression extends Progression {
   protected long prev;

   // Por omissão cria fibonacci clássico: 0, 1, 1, 2, 3, 5, ...
   public FibonacciProgression() { this(0, 1); }

   // Cria fibonacci com dois valores iniciais dados
   public FibonacciProgression(long first, long second) {
      super(first);
      prev = second - first; // valor fictício antes do first
   }

   protected void advance( ) {
      long temp = prev;
      prev = current;
      current += temp;
   }
}
```

Exemplo de heranças: uso

```
public class TestProgression {
   public static void main(String[ ] args) {
      Progression prog;

      System.out.print("Deafault arithmetic progression: ");
      prog = new ArithmeticProgression();
      prog.printProgression(10); // 0 1 2 3 4 5 6 7 8 9

      System.out.print("Arithmetic progression with increment 5: ");
      prog = new ArithmeticProgression(5);
      prog.printProgression(10); // 0 5 10 15 20 25 30 35 40 45

      System.out.print("Arithmetic progression with start 2: ");
      prog = new ArithmeticProgression(5, 2);
      prog.printProgression(10); // 2 7 12 17 22 27 32 37 42 47

      System.out.print("Geometric progression with default base: ");
      prog = new GeometricProgression();
      prog.printProgression(10); // 1 2 4 8 16 32 64 128 256 512

      System.out.print("Geometric progression with base 3: ");
      prog = new GeometricProgression(3);
      prog.printProgression(10); // 1 3 9 27 81 243 729 2187 6561 19683
      // Continua no próximo slide
```

Exemplo de heranças: uso

```
        // Continuação do slide anterior

        System.out.print("Fibonacci with default start values: ");
        prog = new FibonacciProgression( );
        prog.printProgression(10); // 0 1 1 2 3 5 8 13 21 34

        System.out.print("Fibonacci with start values 4 and 6: ");
        prog = new FibonacciProgression(4, 6);
        prog.printProgression(8); // 4 6 10 16 26 42 68 110
    }
}
```