Parte 4.1 – Classes, Coleções

Fernando Barros, Karima Castro, Luís Cordeiro,
Marília Curado, Nuno Pimenta

Os conteúdos desta apresentação baseiam-se nos materiais produzidos por António José Mendes para a unidade curricular de Programação Orientada a Objetos.

Quaisquer erros introduzidos são da inteira responsabilidade dos autores.

Classes e Objetos

- Tal como o nome indica, a programação orientada aos objetos consiste na construção de programas a partir de **objetos**
- Desenhar um programa consiste em definir os **objetos** necessários, a **informação** que contêm, as suas **funcionalidades** e o modo como comunicam entre si, de forma a atingir os objetivos pretendidos
- Um objeto é uma combinação de dados (**variáveis**) e ações (**métodos**) intimamente relacionados
- Um objeto é constituídos por três partes: **identidade**, **atributos** e **comportamentos**

Classes e Objetos

- É possível (e comum) ter objetos semelhantes, com o mesmo comportamento, mas com atributos e identidade diferentes
- A construção de objetos semelhantes é possível mediante a criação prévia de uma **classe**
- Uma classe funciona como um "molde" para a criação de objetos. É usada para definir as características dos objetos criados a partir desta: **atributos** (variáveis) e **comportamentos** (métodos)
- Um objeto é uma instância de uma classe, tendo todas as variáveis e métodos definidos nessa classe.

Classes e Objetos

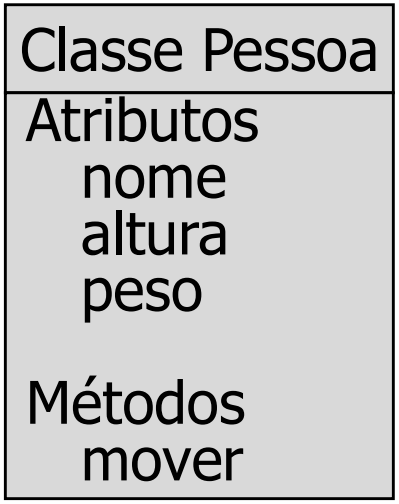
- Por convenção, em Java os nomes das classes começam com maiúscula, bem como a inicial de cada palavra que o constitui (ex: EstaClasse)
- Por sua vez, os nomes dos objetos (variáveis que os referenciam) seguem a mesma regra com exceção da primeira letra que é minúscula (ex: esteObjeto)

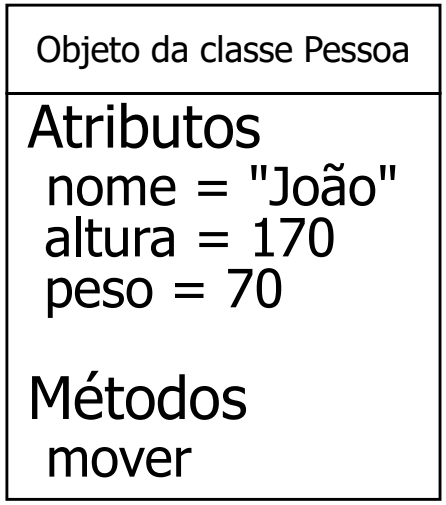
Classes e Objetos

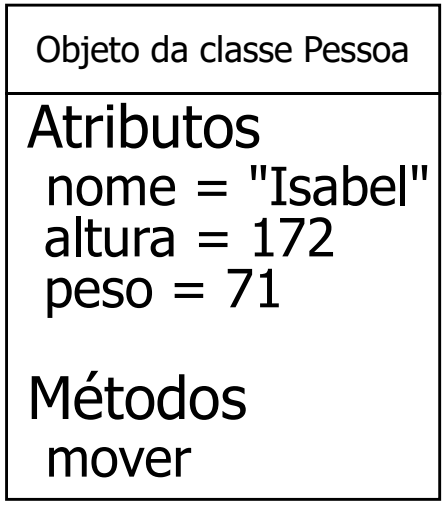
- Objetos
  - Substantivos
  - Coisas reais
- Atributos
  - Propriedades que o objeto tem
- Métodos
  - Ações que o objeto pode fazer
- Mensagens
  - Comunicação entre objetos
  - Um objeto pode pedir a outro para executar um dado método

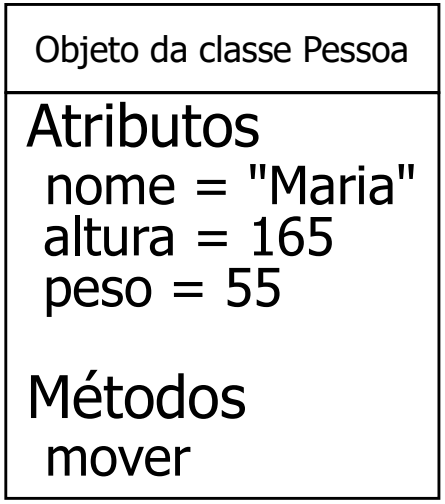
Classes e Objetos

- É possível (e comum) criar vários objetos a partir de uma mesma classe









Classes e Objetos

- A criação de objetos é conseguida através da utilização do operador **new**, seguido do nome da classe a partir da qual se quer criar o objeto e de parêntesis ()
  - Se necessitar de parâmetros estes aparecem entre os ()
  - Exemplo: `new Pessoa("Maria",165, 55);`
- Os objetos são criados na memória central
- Quando um objeto é criado é reservado um bloco de memória suficiente para armazenar todas as variáveis do objeto (nome, altura e peso, no nosso exemplo)
- Este bloco de memória ficará ocupado até que o objeto seja destruído

Classes e Objetos

- A localização de um objeto em memória não é controlada pelo programador
- A interação com um objeto é conseguida através de uma referência
- Uma referência é um tipo especial de valor que identifica um objeto (endereço de memória onde este se encontra)
- As referências podem ser armazenadas em variáveis, por exemplo:

```
Pessoa at;
at = new Pessoa("Maria", 165, 55);
```

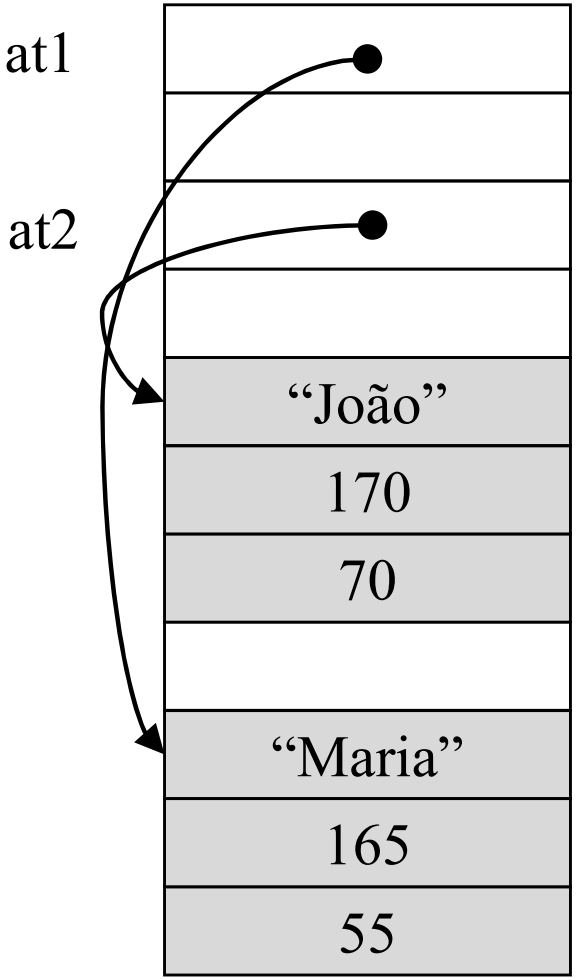
Classes e Objetos

- Após a sua declaração, sabe-se que a variável **at** referencia objeto(s) da classe **Pessoa**, mas o seu valor é indefinido
- A variável só passa a ter um valor definido após a utilização de **new**, por isso é comum usar-se apenas:

```
Pessoa at = new Pessoa("Maria", 165, 55);
```

- A variável **at** passa a armazenar a referência do objeto criado
- Podemos interagir com o objeto através desta variável

Classes e Objetos



- É importante notar a diferença entre a **referência** a um objeto (indicação do local/endereço de memória onde o objeto se encontra) e o próprio **objeto**

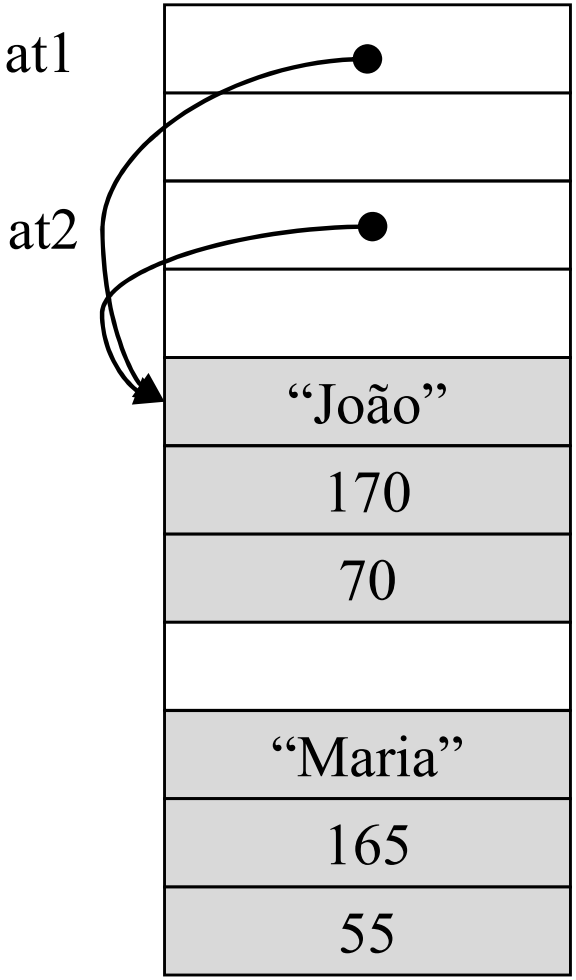
- Por exemplo, o seguinte código pode dar origem ao cenário apresentado na figura

```
Pessoa at1= new Pessoa("Maria",165,55);
Pessoa at2= new Pessoa("João",170,70);
```

- Neste sentido, qual o resultado da seguinte operação?

```
at1 = at2;
```

Classes e Objetos



- Resultado da seguinte operação:

```
at1 = at2;
```

- Os objetos (conteúdo) permanecem inalterados
- A variável **at1** passa a referenciar o mesmo objeto que a variável **at2**
- Perde-se a referência ao objeto inicialmente referenciado por **at1**, passando a estar **inacessível**
- Na verdade, é copiado o endereço de memória que está em **at2** para **at1**

Classes e Objetos

- Quando um objeto já não tem qualquer referência válida para si, não pode ser acedido pelo programa
- É inútil e, por isso, chamado lixo ou **garbage**
- O Java efetua periodicamente recolha de lixo (**garbage collection**), devolvendo a memória ocupada por estes objetos ao sistema, de modo a que possa voltar a ser utilizada
- Noutras linguagens (ex.: C) é o programador que tem que se preocupar em fazer a **garbage collection** (ex: `free(pt);`)

Classes e Objetos

- Como já foi referido, a definição de uma classe implica a especificação dos seus atributos (variáveis) e dos seus comportamentos (métodos)
- Cada objeto criado a partir de uma dada classe tem também esses atributos e comportamentos
- Para conseguir que um objeto mostre um dado comportamento (execute um dado método) é necessário enviar-lhe uma **mensagem**
- Uma mensagem é um pedido que se faz a um objeto para que apresente um dado comportamento (o objeto terá que ser instância de uma classe com esse comportamento definido)

Classes e Objetos

- Para enviar uma mensagem a um objeto é necessária uma instrução que consiste em:
  - Uma referência ao objeto recetor (ex: `System.out`)
  - Um ponto
  - A mensagem que se pretende enviar (ex: `println`)
- Exemplo: `System.out.println("Boa tarde…");`

Classes e Objetos

- O Java Development Kit (JDK) fornece centenas de classes pré-definidas para representar objetos de utilização comum
- O programador pode definir novas classes à medida das suas necessidades (como veremos mais tarde)
- A biblioteca de classes fornecida com o Java é conhecida como **Java API**
- As classes do Java API estão divididas em **packages** que agrupam classes de algum modo relacionadas
- Algumas packages comuns: **java.lang, java.util, java.awt, …**

Classes e Objetos

- Para criar um objeto de uma classe incluída no Java API podemos utilizar o seu nome completo. Por exemplo:

```
java.util.Random random = new java.util.Random();
```

- Em alternativa podemos começar por importar a classe utilizando a instrução **import** e, posteriormente, criar o objeto:

```
import java.util.Random;
…
Random random = new Random();
random.nextInt(100); //{0, …, 99}
```

Classes e Objetos

- Também é possível importar todas as classes de uma dada package. O exemplo seguinte importa todas as classes da package java.util:

```
import java.util.*;
```

- A package **java.lang** é automaticamente importada para todos os programas em Java, pelo que não é necessário efetuar a sua importação explícita

Criação de Classes

- Para além de utilizar classes pré-definidas, num projeto é normal que seja necessário definir novas classes que ajudem a atingir o fim pretendido
- A sintaxe para definir uma classe é a seguinte:

```
class NomeDaClasse {
    declarações de atributos
    construtores
    métodos
}
```

- As variáveis, construtores e métodos de uma classe são genericamente chamados membros dessa classe

Criação de Classes

- Considere-se que se pretendia um programa que leia os dados de um conjunto de estudantes (nome e um conjunto de notas), calcule média obtida por cada estudante e os ordene por ordem decrescente das médias
- O primeiro passo será definir uma nova classe, a classe **Estudante**, para representar um estudante, que terá como atributos o nome e as notas, e como comportamentos a inicialização, o cálculo e obtenção da média e a escrita dos seus dados
- Terá que haver uma outra classe capaz de guardar o conjunto de estudantes e ordená-los de acordo com a sua média

Criação de Classes

- Criação da classe Estudante

```
class Estudante {
  //Atributos
  private String nome;
  private int[] notas;

  //Construtor da classe, promove a inicialização dos
atributos
  public Estudante() {
      ...
  }
  //Comportamentos
  ...
}
```

Criação de Classes

- Construtores
  - Cada classe tem pelo menos um construtor
  - É um tipo especial de método utilizado na criação de objetos dessa classe
  - Muitas vezes são usados para inicializar as variáveis (atributos)
  - Têm o mesmo nome da classe
  - Não têm valor de retorno, nem mesmo void
  - Não podem ser invocados como os restantes métodos
  - O que distingue diferentes construtores da mesma classe é a lista dos seus argumentos

Criação de Classes

- Os construtores são invocados pelo operador **new**
- Assim, **new String("Olá");** implica a invocação a um construtor da classe String que recebe uma cadeia de caracteres como argumento
- A criação de um novo objeto da classe Estudante pode ser conseguida por:

```
Estudante est = new Estudante();
```

Criação de Classes

- Possível implementação do construtor da classe Estudante:

```
public Estudante() {
  System.out.print("Nome do estudante: ");
  Scanner sc = new Scanner(System.in);
  nome = sc.nextLine();
  System.out.print("Quantas notas? ");
  int numNotas = sc.nextInt();
  //Cria o vetor notas com a dimensão necessária
  notas = new int[numNotas];
  for (int i = 0; i < numNotas; i++) {
     System.out.printf("Nota (%d) do aluno: ", i + 1);
     notas[i] = sc.nextInt();
  }
}
```

Criação de Classes

- Para evitar que os construtores interajam com o utilizador (recomendado para diferentes interfaces, por exemplo), estes devem receber os dados necessários como argumentos. Exemplo:

```
public Estudante(String nomeEst, int[] notasEst) {
  // Cria um objeto nome com a string recebida
  nome = nomeEst;
  //Cria o vetor notas com a dimensão necessária
  notas = new int[notasEst.length];
  for (int i = 0; i < notasEst.length; i++) {
     notas[i] = notasEst[i];
  }
}
```

Criação de Classes

- Possível implementação dos comportamentos/métodos:

```
//Método para cálculo da média
private float calculaMedia() {
    float soma = 0.;
    if (notas.length > 0) {
        for (int n: notas)
           soma += n;
        return soma / notas.length;
    } else return -1;
}
```

Criação de Classes

- Possível implementação dos comportamentos/métodos:

```
//Método para cálculo da média arredondada
public int media() {
    return Math.round(calculaMedia());
}
```

Criação de Classes

```
//Método de acesso externo à média
public float getMedia() {
    return calculaMedia();
}
//Escreve os dados de um estudante
public void imprimeEstudante() {
    System.out.print("As notas de "+ nome+ " são: ");
    for (int n: notas)
        System.out.print(n + " ");
    System.out.println();
    System.out.println("A média é "+calculaMedia());
}
```

Criação de Classes

- Possível implementação do programa principal:

```
//Possível implementação do método main()
public static void main(String[] args) {
   Estudante est = new Estudante();
   float media = est.getMedia();
   System.out.println("A média é " + media);
   est.imprimeEstudante();
}
```

Criação de Classes

- Os atributos (nome e notas neste exemplo) chamam-se **variáveis de instância** (instance variables) porque pertencem ao objeto como um todo e não a um método em particular
- As declarações de **variáveis de instância** começam geralmente com a palavra **private**, significando que são privativas do objeto, não sendo acessíveis diretamente do seu exterior (por outro objeto, mesmo que seja do mesmo tipo / classe)

Criação de Classes

- Uma classe define o tipo de dados para um objeto, mas não armazena valores
- Cada objeto tem o seu único espaço de dados em memória, onde pode guardar os seus valores
- Todos os métodos de uma classe têm acesso às variáveis de instância dessa classe
- Os métodos de uma classe são partilhados por todos os objetos dessa classe
- Quando chamado a partir de um determinado objeto, um método tem acesso aos valores das variáveis de instância desse objeto, e não de outros objetos (ainda que da mesma classe)

Encapsulamento

- Visto do exterior, um objeto é uma entidade **encapsulada**, fornecendo um conjunto de serviços
- Estes serviços são conseguidos à custa dos métodos públicos do objeto
- Ao conjunto de serviços fornecidos por um objeto chama-se **interface** desse objeto
- Um objeto deve ser **autocontido**, ou seja qualquer alteração do seu estado (das suas variáveis) deve ser provocada apenas pelos seus métodos
- **Um objeto não deve permitir que outro objeto altere o seu estado**

Encapsulamento

- O cliente de um objeto deve poder requisitar os seus serviços, mas sem saber como isso será conseguido
- O encapsulamento consegue-se à custa da utilização dos **modificadores de visibilidade** disponíveis na linguagem
- Em Java há três modificadores de visibilidade: **public**, **private** e **protected**
- O modificador **protected** será visto um pouco mais tarde
- Os membros de uma classe que forem declarados com o modificador **public** podem ser acedidos a partir de qualquer ponto (incluindo de outros objetos, mesmo que não sejam da mesma classe)

Encapsulamento

- Os membros declarados com **private** só podem ser acedidos de dentro do objeto
- Os membros declarados **sem qualquer modificador** têm visibilidade por defeito e podem ser acedidas a partir de qualquer classe dentro da mesma package
- De uma forma geral, as **variáveis de instância** dos objetos não devem ser declaradas com visibilidade public (antes com **private**)
- Os métodos que implementam os serviços fornecidos pelo objeto são declarados como public, por forma a poderem ser chamados a partir de outros objetos

Encapsulamento

- Os métodos públicos chamam-se **serviços**
- Os restantes métodos são **métodos de suporte**, servem como auxiliares dos serviços e não devem ser declarados como públicos

Variáveis e métodos de classe

- Em Java existe o modificador **static** que pode ser aplicado a variáveis ou métodos
- Serve para associar a variável ou método à classe e não a cada objeto da classe
- Se uma variável é declarada como **static**, não é replicada mas sim partilhada por todos os objetos dessa classe
- Isto significa que alterar essa variável num objeto, provoca a sua alteração também nos restantes objetos dessa classe (só existe uma variável em memória)
- Estas variáveis chamam-se **variáveis de classe**

Variáveis e métodos de classe

```
class Estudante {
  //Atributos
  ...
  //Para contar o número de Objetos instanciados
  private static int conta = 0;
  //Construtor da classe
        ...
  //Comportamentos
  ...
  public static int getConta() {return conta;}
  public Estudante() {
  //Incrementa a Variável de Classe por cada novo Objeto
     conta++;
     ...
  }
}
```

Variáveis e métodos de classe

- Normalmente chamamos um método através de uma instância da classe (objeto)
- Se um método é declarado como **static** pode ser chamado através do nome da classe, não sendo necessário que exista um objeto dessa classe
- Por exemplo, a classe Math tem vários métodos estáticos que podem ser chamados sem que seja criado um objeto desta classe:
  - **Math.abs(num)** - valor absoluto
  - **Math.sqrt(num)** - raiz quadrada
  - **Math.pow(x, y)** - potência

Variáveis e métodos de classe

- O método **main** é estático. É chamado a partir do sistema sem que este tenha que criar um objeto
- Os métodos estáticos não podem aceder a **instance variables**, uma vez que estas apenas existem nos objetos
- Podem utilizar variáveis declaradas com **static** (variáveis de classe) e também variáveis locais ao método
- Estes métodos são normalmente chamados **métodos de classe**

Keyword **this**

- A keyword **this** pode ser utilizada em **métodos** e **construtores**
- Representa um **handle** ou **referência** para o próprio objeto que está a executar o código
- **Exemplo 1:** um método tem um argumento com o mesmo nome de um atributo da classe

```
public Estudante(String nome, int[] notasEst) {
  // Cria um objeto nome com a string recebida
  this.nome = nome;
  ...
}
```

Atributo da classe, precedido de **this.**

Argumento do método

- **this.** Distingue argumento do atributo com o mesmo nome

Keyword **this**

- **Exemplo 2:** um construtor "chama"/recorre outro construtor da mesma classe (só pode ser feito uma vez, no início do código do construtor)

```
public Estudante(String nome) {
  // Cria um objeto nome com a string recebida
  this.nome = nome;
  ...
}
public Estudante(String nome, int[] notasEst) {
  this(nome);
  ...
}
```

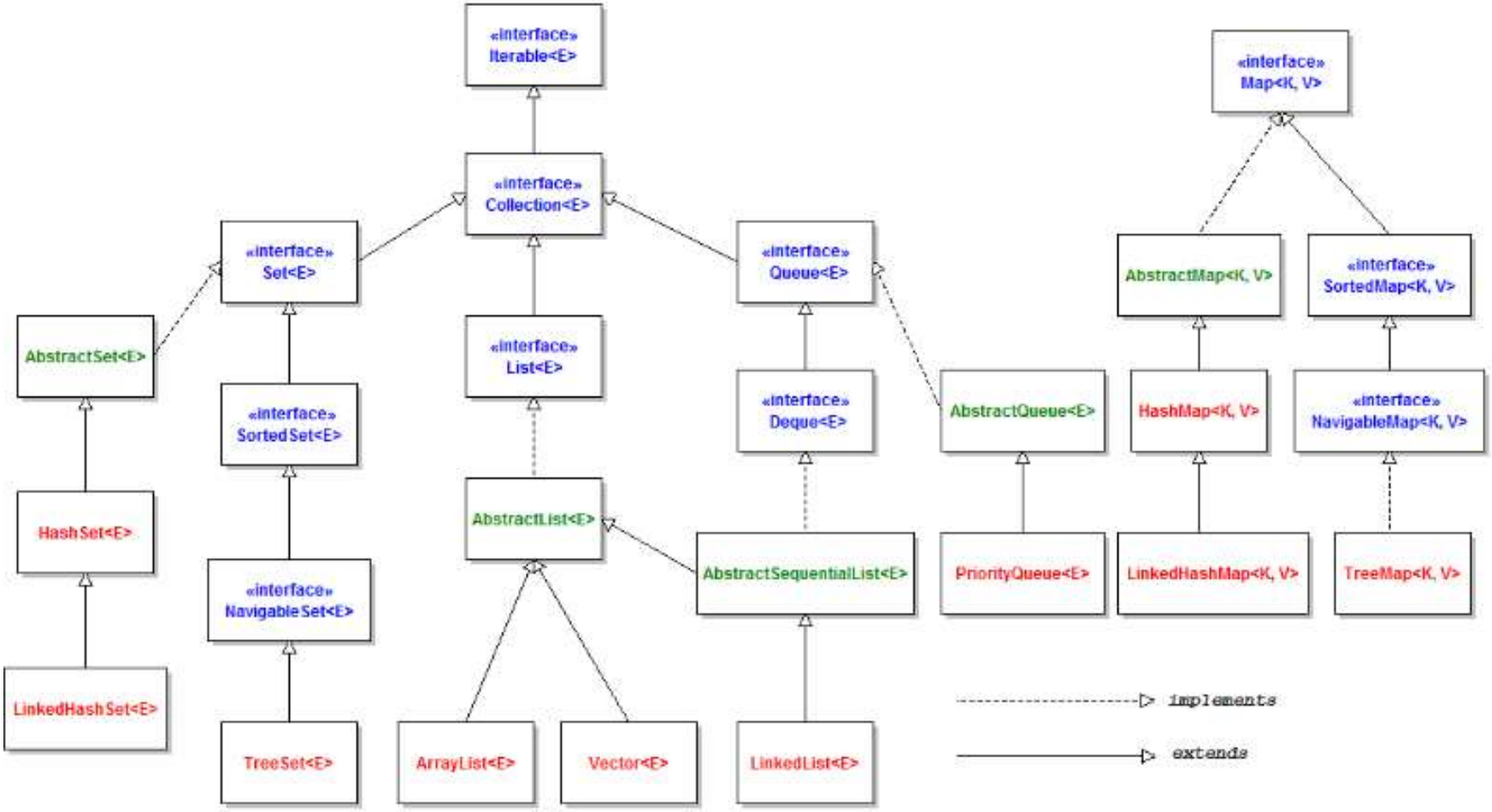
this seguido de () **representa o construtor**, da mesma classe, com uma lista de argumentos correspondente aos parâmetros fornecidos entre ()

COLEÇÕES

Coleções

- Uma coleção é um objeto capaz de agrupar múltiplos elementos numa única unidade de representação e processamento, possuindo propriedades de estrutura e funcionamento próprias
- A ***Java Collections Framework*** é uma arquitetura unificada, constituída por interfaces, classes abstractas, classes concretas e métodos que visa representar e manipular coleções
  - A JCF contém, entre outras coisas, a *Collections Interface*, as APIs fundamentais que estruturam todas as classes

Coleções



Java Collections Framework (JCF)

https://www.codejava.net/java-core/collections/overview-of-java-collections-framework-api-uml-diagram

Coleções

- A JCF utiliza 4 interfaces principais:
  - `Set`: conjunto de objetos, não existindo a noção de posição (primeiro, último, …), nem sendo permitidos duplicados
  - `List`: sequência de objetos, existindo a noção de ordem e permitindo duplicados
  - `Map`: correspondências unívocas entre objetos (chave – valor), em que as chaves são um conjunto, logo sem elementos repetidos
  - `Queue`: estruturas lineares do tipo FIFO

Coleções

- A verificação de tipos é feita em tempo de compilação
- Manipulação de colecções:
  - **Auto-Boxing e Auto-Unboxing**: conversão automática de valores dos tipos primitivos (`int`, …) em objetos das classes correspondentes (`Integer`, …) quando são inseridos em coleções e a conversão contrária quando são lidos de coleções
  - **Iterador foreach**: ciclo for aplicado a coleções que podem ser iteradas, substituindo os iteradores explícitos
  - **Tipos genéricos**: adicionam às coleções generalidade e segurança em tempo de compilação

Tipos genéricos

- Um **tipo genérico** é um tipo referenciado (classe ou interface) que usa na sua definição um ou mais tipos de dados (não primitivos) como parâmetro, que serão mais tarde substituídos por tipos concretos quando o tipo genérico for instanciado
  - Por exemplo, `Stack<E>` representa uma pilha de “alguma coisa”, sendo que esta pode ser um qualquer tipo não primitivo

Tipos genéricos

- Considermos a classe Ponto para representar pontos numa geometria 2D usando diferentes graus de precisão

```
class Ponto<E> {
    private E x, y;
    public Ponto(E x, E y) {
        this.x = x;
        this.y = y;
    }
    public E x() {return x;}
    public E y() {return y;}
    public void x(E x) {this.x = x;}
    public void y(E y) {this.y = y;}
    public static void test() {
        Ponto<Integer> pixel = new Ponto<>(4, 6);
        Ponto<Double> localizacao = new Ponto<>(4.0, 6.5);
    }
}
```

Tipos genéricos

- Um **tipo parametrizado** é o resultado de uma instanciação de um tipo genérico para um valor concreto da variável de tipo E
  - Por exemplo `Stack<String>` ou `Stack<Estudante>`
- O tipo parametrizado resultante da instanciação pode ser usado em declaração de variáveis, parâmetros e resultados de métodos
- As coleções beneficiaram da introdução de tipos genéricos, uma vez que é possível fazer a sua programação em função de um tipo genérico de dados a armazenar, o qual é substituído por um tipo concreto no momento da instanciação

Tipos genéricos

- Existem várias classes que implementam `List<E>` entre as quais:
  - `ArrayList<E>`
  - `Vector<E>`
  - `Stack<E>` (subclasse de `Vector<E>`)
  - `LinkedList<E>`

Tipos genéricos

- As classes que implementam `List<E>` devem satisfazer o seguinte:
  - Existe uma ordem nos seus elementos (é uma lista de objetos de tipo `E`)
  - Cada elemento ocupa uma posição referenciada por um índice inteiro a partir de 0
  - Os elementos podem ser inseridos em qualquer ponto da lista
  - A lista pode conter elementos em duplicado e elementos `null`

Classe ArrayList<E>

- A classe `ArrayList<E>` implementa listas utilizando um array dinâmico, ou seja redimensionável em execução
- Para criar um `ArrayList` é necessário definir qual o tipo dos seus elementos (classe, classe abstracta ou interface, já que nenhuma coleção armazena valores primitivos)
  - Por exemplo, se quisermos guardar uma lista de nomes devemos ter E = String, pelo que a criação de uma `ArrayList<String>` será:

```
List<String> amigos = new ArrayList<>(); ou
List<String> amigos = new ArrayList<>(50);
```

Classe ArrayList<E>

- Sobre esta nova estrutura é possível fazer as operações definidas na interface `List<E>`, por exemplo:

```
int tam1 = amigos.size();
amigos.add("João");
String nomeAmigo = amigos.get(0);
```

- A instrução `amigos.add(new Integer(127));` geraria um erro de compilação, pois o compilador sabe que `amigos` é um `ArrayList<String>`, pelo que não pode aceitar elementos de outros tipos (ainda que objetos)

Classe ArrayList<E>: API

Método	Funcionalidade
`void add(E o)`	Adiciona o objeto o (do tipo E) no fim do ArrayList
`add(int indice, E o)`	Insere o objeto o na posição do ArrayList indicada por indice
`void remove(int indice)`	Remove o objeto armazenado na posição dada por indice
`void clear()`	Elimina todos os elementos do ArrayList
`int indexOf(E o)`	Devolve o índice da primeira posição onde se encontra um objeto igual a o ou –1 se não o encontrar
`E get(int indice)`	Devolve o elemento que ocupa a posição definida por indice
`void set(int indice, E o)`	Substitui o elemento da posição dada por indice pelo objeto o
`List<E> subList(int a, int b)`	Devolve um ArrayList formado pelos elementos colocados entre as posições a e b-1
`boolean isEmpty()`	Indica se o ArrayList está vazio
`int size()`	Devolve o número de elementos do ArrayList

Iteradores

- A iteração sobre coleções pode ser feita da maneira clássica usando índices:

```
for (int i = 0; i < amigos.size(); ++i)
  System.out.println("Amigo: " + amigos.get(i));
```

- No entanto, existe a interface `Iterator<E>` (superinterface de `Collection<E>`) que obriga à existência de um método `iterator();` que cria um objeto `Iterator<E>`, que é um iterador automático sobre uma coleção de elementos de tipo E
- Este trabalha sobre a coleção usando 3 métodos:
  - `hasNext(); next(); remove();`

Iteradores

- Uma das formas de usar um iterador:

```
Iterator<String> it = amigos.iterator();
while (it.hasNext())
  System.out.println("Amigo: " + it.next());
```

  - `it.next()` devolve uma `String`, pois é esse o tipo dos elementos do `ArrayList` e do `Iterator`
- Iteração sobre colecções usando uma forma alternativa de ciclo for
  - A sua forma genérica é:

```
for (Tipo elem: coleçãoIterável<Tipo>)
     instruções
```

Iteradores

- O que no nosso exemplo dará:

```
for (String nome: amigos)
  System.out.println("Amigo: " + nome);
```

- A sua utilização só faz sentido quando se quer iterar sobre toda a colecção
- Por exemplo, num algoritmo de pesquisa isso não faz sentido, pois só interessa procurar até encontrar o elemento pretendido
  - Nesse caso deve usar-se o `while` associado a um iterador, incluindo a expressão de pesquisa na sua condição

```
while (it.hasNext() && ! encontrado) ...
```

Exemplo

- Pretende-se um programa que leia os dados de um conjunto de estudantes (nome e um conjunto de notas), calcule a sua média e imprima os dados dos estudantes com maior média
  - Classe `Estudante` para representar um estudante
  - Classe para representar uma turma de estudantes (`Turma`)
    - Vai guardar um conjunto de estudantes num `ArrayList`
    - Tem como comportamentos:
      - a sua inicialização
      - a adição de um estudante ao `ArrayList<Estudante>`
      - a escrita dos dados de todos os estudantes
      - a escrita dos estudante com melhor média
  - Classe para fazer a gestão do sistema (`GereTurma`)

Exemplo

```
class Turma {
 private List<Estudante> estudantes;
 public Turma() {
      estudantes = new ArrayList<Estudante>();
 }
 //Adiciona um novo estudante
 public void juntaEstudante() {
    Estudante e = new Estudante();
     estudantes.add(e);
 }
 //Imprime os dados de todos os estudantes
 public void imprimeTurma() {
    for (Estudante e: estudantes)
       System.out.println(e);
 }
}
```

Exemplo

```
//Imprime os estudantes com maior média
public void imprimeMaiores() {
    if (estudantes.isEmpty())
        return;
    Estudante maior = estudantes.get(0);
    ArrayList<Estudante> maiores = new ArrayList<>();
    maiores.add(maior);
    for (int i = 1; i < estudantes.size(); ++i) {
        Estudante e = estudantes.get(i);
        if (e.media() < maior.media())
            continue;
        if (e.media() > maior.media()) {
            maiores.clear();
            maior = e;
        }
        maiores.add(e);
    }
    for (Estudante e: maiores)
        System.out.println(e);
}
```

Exemplo

```
class GereTurma {
  public static void main(String[] args) {
      Turma t = new Turma(); // Cria uma turma
      int escolha;
      Scanner stdin = new Scanner(System.in);
      do {// Menu
          System.out.println("1 - Adicionar estudante");
          System.out.println("2 - Lista de estudantes");
          System.out.println("3 - Melhor Média");
          System.out.println("0 - Sair");
          escolha = stdin.nextInt();
          switch (escolha) {
              case 1: t.juntaEstudante(); break;
              case 2: t.imprimeTurma(); break;
              case 3: t.imprimeMaiores(); break;
              case 0: System.exit(0);
          }
      } while (escolha != 0);
      stdin.close();
  }
}
```

Iteradores

- As listas (e só estas) têm também um método `listIterator()` que devolve um iterador especial `ListIterator<E>`, que acrescenta métodos que possibilitam:
  - Navegar na lista em sentido contrário, com `hasPrevious()` e `previous()`
  - Saber o índice do próximo elemento a ser iterado em qualquer dos sentidos, `nextIndex()` e `previousIndex()`
  - Fazer `remove()`, `set(E e)` e `add(E e)` em qualquer momento da iteração
    - `remove()` e `set(E e)`, removem e alteram o último elemento iterado, respectivamente

Iteradores

- Alguns dos métodos de `ArrayList` usam iteradores para implementar métodos que trabalham com grande quantidade de dados. Por exemplo:
  - `addAll(coleção)`: junta ao fim da lista receptora os elementos da coleção fornecida como parâmetro, pela ordem dada pelo iterador desta (as colecções devem ser compatíveis)
  - `removeAll(coleção)`: remove do receptor os elementos da lista fornecida como parâmetro
  - `retainAll(coleção)`: remove do receptor os elementos que não pertencem à lista fornecida como parâmetro
  - `containsAll(coleção)`: verifica se todos os elementos da lista fornecida como parâmetro pertencem à lista receptora

Exemplo

- Um exemplo de utilização de iteradores num algoritmo de inversão de um ArrayList:

```
List<String> inverte(List<String> aList) {
 ArrayList<String> bList = new ArrayList<>(aList);
 ListIterator<String> forwardI = bList.listIterator();
 ListIterator<String> reverseI = bList.listIterator(bList.size());
 for (int curr = 0, meio = aList.size()/2; curr < meio; ++curr) {
     String left = forwardI.next();
     String right = reverseI.previous();
     reverseI.set(left);
     forwardI.set(right);
 }
 return bList;
}
```

ENUMERAÇÕES

Enumerações

- Uma enumeração (`enum`) é uma classe especial que permite representar valores constantes
- A *keyword* enum permite representar uma enumeração

```
enum Valor {
  Alto, Médio, Baixo
}
```

- Os valores podem ser acedidos através do operador '.'

```
Valor valor = Valor.Alto;
System.out.print(valor);        //Alto
```

- Pode também iterar-se sobre os valores duma enumeração:

```
for (Valor v: Valor.values())
  System.out.print(v + " ");   // Alto Médio Baixo
```

Enumerações

```
enum Pessoa {
   Bomtempo, Garrett, Newton, Renoir, Seixas, Woolf, Vermeer
}
public static void print(Pessoa pessoa) {
   String título = switch (pessoa) {
        case Vermeer, Renoir: yield "pintor(a)";
        case Bomtempo, Seixas: yield "compositor(a)";
        case Garrett, Woolf: yield "escritor(a)";
        default:
         System.out.printf("Não conheço %s%n", pessoa);
            yield "...";
     };
     System.out.printf("%s foi um(a) %s%n", pessoa, título);
}
```

Enumerações

```
public static void main(String[] args) {
    print(Pessoa.Bomtempo);
    print(Pessoa.Vermeer);
    print(Pessoa.Newton);
}

Bomtempo foi um(a) compositor(a)
Vermeer foi um(a) pintor(a)
Não conheço Newton
Newton foi um(a) …
```