# Programação Orientada por Objectos

# Introdução à Coleções - Sets

Prof. Rui César das Neves, Prof. José Cordeiro

Departamento de Sistemas e Informática

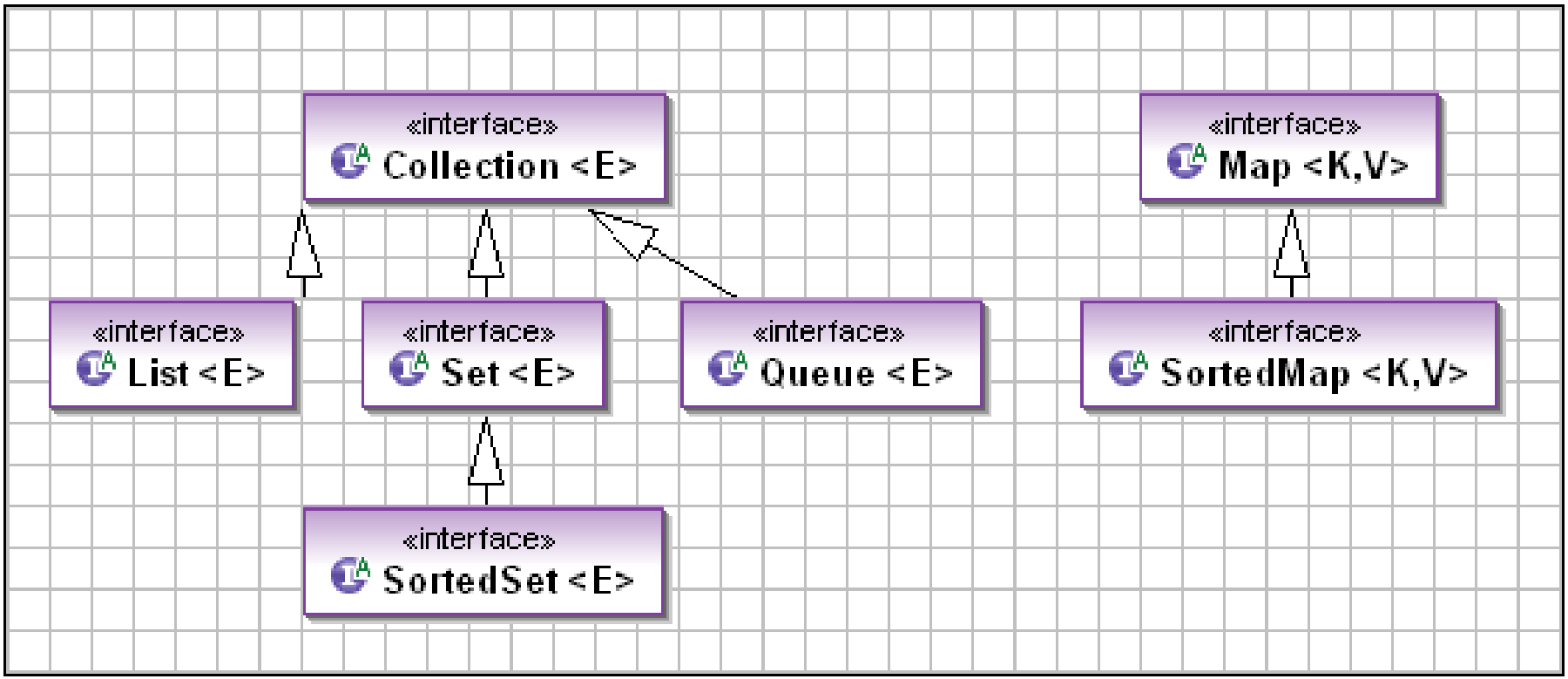
Escola Superior de Tecnologia de Setúbal – Instituto Politécnico de Setúbal

2014/2015

# Sumário

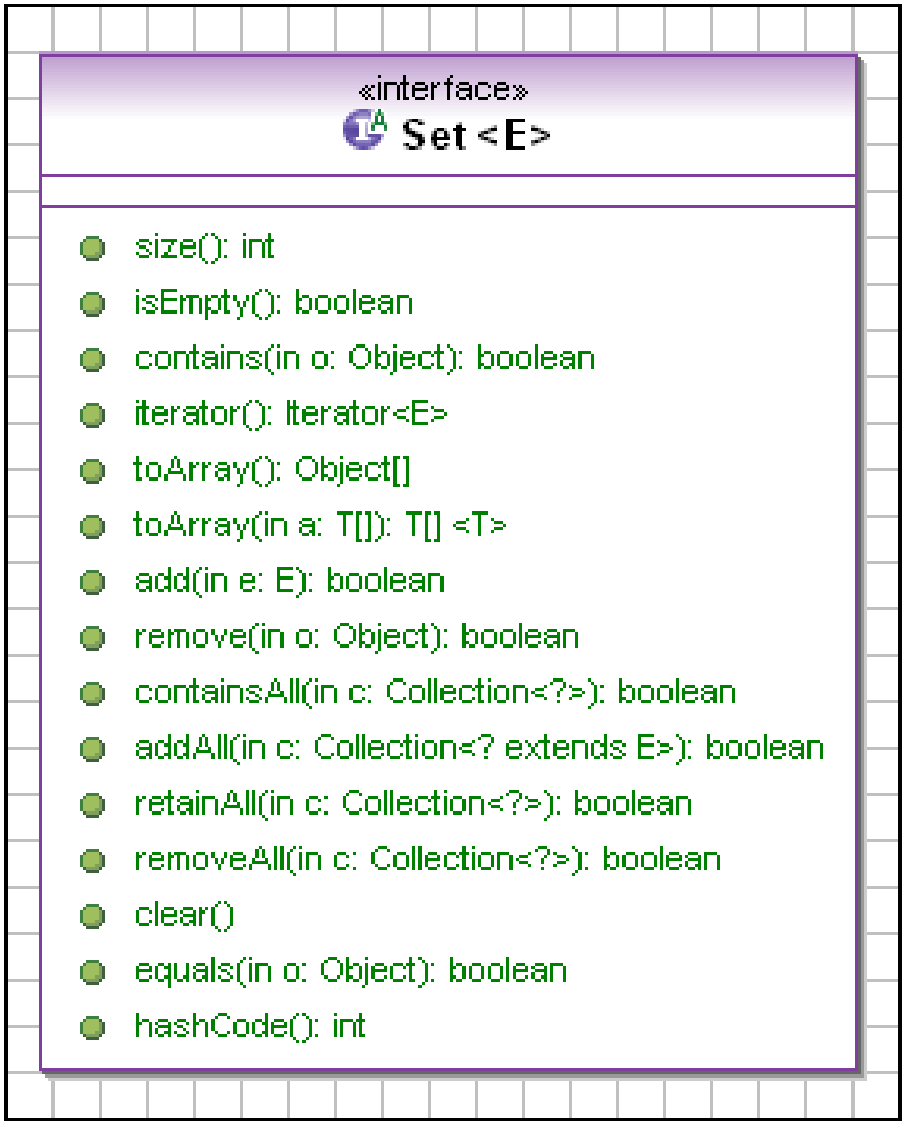
- Conjuntos
  - Interface **`Set`**
    - Classe **`HashSet<E>`**
    - Classe **`TreeSet<E>`**
    - Classe **`LinkedHashSet<E>`**
- Manipulação de conjuntos
- Unicidade da informação
- Métodos **`equals`** e **`hashCode`**

# Java Collections Framework – Interfaces Genéricas



# Coleções – Interface Set

- Interface que impõe a não existência de duplicações na coleção.
  - Representa a noção matemática de conjuntos.
  - Não existe ordenação de qualquer tipo sobre os elementos inseridos.
    - Pelo que ao programar não podemos assumir nada quanto a uma eventual ordem dos elementos!



# Coleções – Classes Set

- Interface **`Set`**:
  - **`HashSet`** – armazena os elementos numa **`hash table`** (explicado mais à frente).
    - É a implementação com melhor desempenho mas não garante nada quanto à ordem de iteração.
    - É a única implementação de conjuntos que estudaremos detalhadamente.
    - **`HashSet<Pessoa> pessoas = new HashSet<>();`**
      - Declara e cria um **`HashSet`** chamado **`pessoas`** para armazenar objetos da classe **`Pessoa`**.
  - Outras implementações de conjuntos (que mantêm a ordem de iteração):
    - **`TreeSet`**
    - **`LinkedHashSet`**

# Utilização de Sets

- A interface **`Set`** não associa a cada elemento uma posição dentro da coleção, como acontece na interface **`List`**.
- Assim a melhor forma para percorrer os elementos de um conjunto é através do ciclo **`For-Each`** ou de um **`iterator`**:

```
for (E elemento : conjunto) {
. . .
}
```

- Atendendo às diferentes implementações possíveis para a interface **`Set`** (**`HashSet`**, **`TreeSet`**, **`LinkedHashSet`**) devemos declarar a variável utilizando essa interface e defini-la escolhendo uma implementação concreta (abordagem que permite minimizar o "Acoplamento de Subclasses", já utilizada nas **`List`**):

```
Set<String> conjunto = new HashSet<>();
```

- Desta forma podemos optar por diferentes implementações (conseguindo ganhos de *performance*) com alterações mínimas.

# Coleções - Classe HashSet

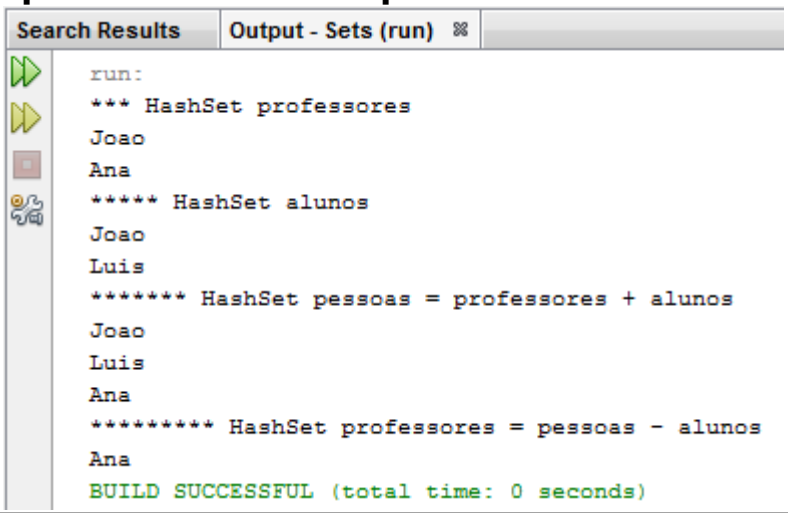
```
public static void main(String[] args) {
  System.out.println("*** HashSet professores");
  Set<String> professores = new HashSet<>();
  professores.add("Ana");
  professores.add("Joao");
  for(String s: professores) {
    System.out.println(s);
  }

  Set<String> alunos = new HashSet<>();
  alunos.add("Joao");
  alunos.add("Luis");
  System.out.println("***** HashSet alunos");
  for(String s: alunos) {
    System.out.println(s);
  }
```

# Coleções - Classe HashSet

```
  Set<String> pessoas = new HashSet<>(professores);
  pessoas.addAll(alunos);
  System.out.println("******* HashSet pessoas = professores + alunos");
  for(String s: pessoas) {
    System.out.println(s);
  }

  professores = new HashSet<>(pessoas);
  professores.removeAll(alunos);
  System.out.println("********* HashSet professores = pessoas -
    alunos");
  for(String s: professores) {
    System.out.println(s);
  }
}
```

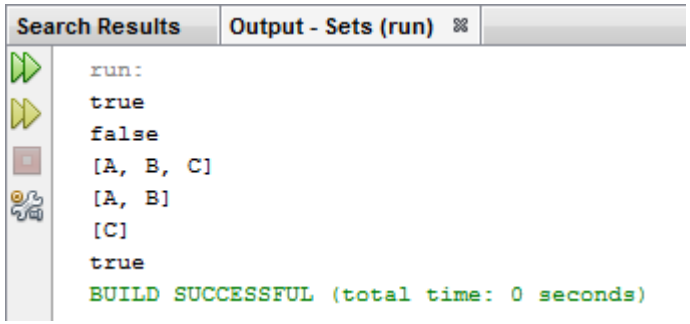


# Operações de Conjuntos

- As tradicionais operações sobre conjuntos (pertença $\in$, união $\cup$, interseção $\cap$, diferença – e contido $\subset$) são feitas com base nos métodos disponibilizados pela interface **`Collection`**:
  - pertença – **`contains`**
  - união – **`addAll`**
  - interseção – **`retainAll`**
  - diferença – **`removeAll`**
  - contido – **`containsAll`**
- Caso se deseje obter o conjunto resultante, sem alterar nenhum dos conjuntos envolvidos, é comum recorrer à criação de um novo conjunto, utilizando o construtor que recebe uma **`Collection`** como argumento, e depois aplicar a operação.

# Operações de Conjuntos

```
public static void main(String[] args) {
  Set<String> c1 = new HashSet<>();
  c1.add("A");c1.add("B"); // c1 = { A, B }
  Set<String> c2 = new HashSet<>();
  c2.add("A");c2.add("B");c2.add("C"); // c2 = { A, B, C }
  System.out.println(c1.contains("A"));
  System.out.println(c1.contains("C"));
  Set<String> uniao = new HashSet<>(c1);
  uniao.addAll(c2);
  System.out.println(uniao);
  Set<String> intersecao = new HashSet<>(c1);
  intersecao.retainAll(c2);
  System.out.println(intersecao);
  Set<String> diferenca = new HashSet<>(c2);
  diferenca.removeAll(c1);
  System.out.println(diferenca);
  System.out.println(c2.containsAll(c1));
}
```



# Unicidade dos Elementos

- Criar um conjunto de elementos da classe **Pessoa** (exemplo simplificado, contendo apenas o atributo **nome** e sem gets ou sets):

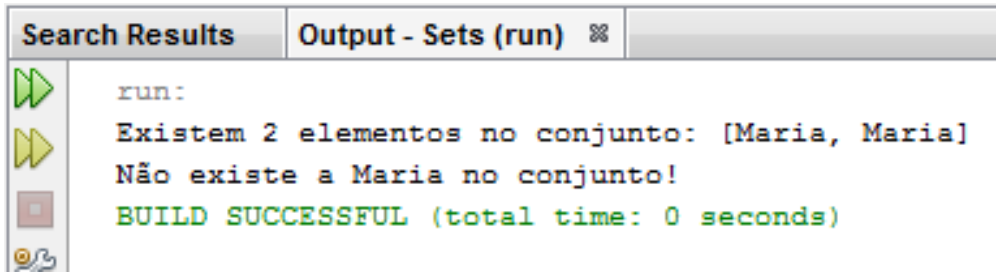
```
public class Pessoa {
  private String nome;

  public Pessoa(String nome) {
    this.nome = nome;
  }

  @Override
  public String toString() {
    return nome;
  }
}
```

# Unicidade dos Elementos

```
public static void main(String[] args) {
  Set<Pessoa> pessoas = new HashSet<>();
  pessoas.add(new Pessoa("Maria"));
  pessoas.add(new Pessoa("Maria"));
  System.out.format("Existem %d elementos no conjunto: %s\n",
                                          pessoas.size(), pessoas);
  if (pessoas.contains(new Pessoa("Maria"))) {
      System.out.println("Existe a Maria no conjunto!");
  } else {
      System.out.println("Não existe a Maria no conjunto!");
  }
}
```



# Unicidade dos Elementos

- A unicidade dos elementos é feita através da chamada ao método **`equals`**.
- Apesar das pessoas envolvidas terem todas o mesmo nome, correspondem a objetos diferentes e por isso o **`equals`** devolve sempre **`false`**, nunca detetando que estamos na presença da "mesma pessoa".
- A solução passa por redefinir o método **`equals`** na classe **`Pessoa`**. Fazendo com que este apenas compare o atributo **`nome`**.

# Redefinir o método equals

```
@Override
public boolean equals(Object obj) {
  if (obj == null) {
      return false;
  }
  if (getClass() != obj.getClass()) {
      return false;
  }
  final Pessoa other = (Pessoa) obj;
  if (!Objects.equals(this.nome, other.nome)) {
      return false;
  }
  return true;
}
```

# Redefinir o método equals

- Redefinir o método **`equals`** não basta: se executarmos o programa continuamos a ter 2 elementos no conjunto e a não conseguir detetar a presença da Maria.
- Este problema está relacionado com o conceito de *hashing* (**`HashSet`**) e que é implementado em cada objeto através do método **`hashCode()`**.
- Por questões de eficiência, um **`HashSet`** não é implementado através de uma sequência de elementos, pois, dessa forma, seria necessário percorrer a sequência sempre que se introduzia um novo elemento para verificar se ele não seria repetido.

# Implementação através de hash table

- Um **`HashSet`** é implementado através de uma **`hash table`**. Uma **`hash table`** pode ser vista como um **`array`** onde os elementos são colocados na posição que se obtém pela chamada ao método **`hashCode`** (implementação hipotética do método que adiciona elementos ao conjunto):

```
public void add (Pessoa pessoa){
  valores[pessoa.hashCode()] = pessoa;
}
```

- O método **`int hashCode()`** existe na classe **`Object`** e tenta devolver, para todos os objetos, um valor que o identifique univocamente.
- Com esta abordagem a determinação da existência de um elemento no conjunto é muito rápida: basta executar o método **`hashCode`**, no elemento, e verificar se a posição respetiva está ocupada, sem ser necessário percorrer todo o conjunto, comparando elemento a elemento.

# Método hashCode

- O método **hashCode** não garante que é devolvido um valor diferente para cada objeto. Tal seria impossível, pois podem existir mais objetos que os valores disponíveis na gama dos inteiros e o algoritmo a implementar seria tão complexo que não seria eficiente.
- Assume-se que método **hashCode** devolve um valor que seja *aleatoriamente distribuído*, por forma a reduzir a hipótese de repetição (chamada "colisão"). Havendo a certeza que elas vão acabar por existir.
- Assim um **HashSet** é implementado através de uma **hash table** onde, em cada posição não será colocado o elemento mas é colocada uma sequência de elementos (designada por *bucket* - balde) que contêm o mesmo **hashCode**. Desta forma a existência de dois objetos com o mesmo **hashCode** não implica que um vá substituir o outro: ficam ambos colocados na mesma sequência.

# Uso do método hashCode

- Ao se introduzir um elemento num **`HashSet`** o sistema começa por determinar a posição do elemento na **`hash table`** através da chamada ao método **`hashCode`**, percorrendo em seguida a sequência de elementos comparando-os através do método **`equals`**.
- Por esta razão os métodos **`equals`** e **`hashCode`** estão intimamente relacionados, não podendo fazer a redefinição de um sem redefinir o outro (se apenas redefinirmos o **`equals`** o **`hashCode`** continua a indicar posições diferentes na **`hash table`** para os objetos, permitindo repetições).
- Devendo na sua redefinição envolver, em ambos os métodos, os mesmos atributos da classe.

# Redefinição do método hashCode

- Na definição do método **`hashCode`** do Java (`http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html`) é dito que:
  - Durante a uma execução de um aplicação Java, a chamada ao método **`hashCode`**, para o mesmo objeto, deve devolver valores iguais. Tal não é garantido para execução distintas (é preciso ter especial cuidado se forem armazenados **`hashCode`** em bases de dados e posteriormente obtidos ou se tivermos aplicações distribuídas por diferentes máquinas);
  - Se dois objetos são **`equals`** então devem ter o mesmo **`hashCode`**;
  - Não é obrigatório que dois objetos que não sejam **`equals`** tenham **`hashCode`** diferentes. Mas uma boa implementação produz valores tendencialmente distintos para aumentar a performance nas **`hash table`**.

# Estratégias de Implementação do hashCode

- Devemos combinar todos os atributos relevantes (que correspondem à unicidade do objeto) no cálculo do **`hashCode`** (e do **`equals`**).
- Pretende-se que os valores inteiros produzidos sejam aleatoriamente distribuídos. Uma vez que iremos utilizar o valor para determinar a posição nas **`hash table`** e para não termos **`hash table`** muito grandes é desejável que os *lower bits* (que correspondem a menores inteiros) sejam aleatoriamente distribuídos (dado um conjunto de elementos, pretende-se que os **`hashCode`** respetivos tenham, nos seus bits, 50%-50% de probabilidades de serem **`0`** ou **`1`**).
- Pretende-se uma execução rápida do cálculo do **`hashCode`** para não sobrecarregar o sistema (por questões de eficiência, poderá ser necessário guardar num atributo o valor do **`hashCode`** que será apenas calculado na primeira vez que é pedido e devolvido, a partir, do atributo nas vezes seguintes)

# Sugestões de implementação do hashCode

- Se os atributos envolvidos forem dois valores inteiros (**x** e **y**) aleatoriamente distribuídos, o **hashCode** pode ser calculado através de **`x ^ y`** (onde **^** é o operador *ou-exclusivo* em binário: devolve **1** se os bits forem diferentes e **0** se forem iguais).
- Se os dois valores inteiros (**x** e **y**) não forem aleatoriamente distribuídos podemos modificar um deles por forma a "quebrar" a má distribuição. Por exemplo, podemos multiplicar o **x** por um valor primo **p** próximo de uma potência de dois§ (ex. 17 ou 33): **`(x * p) ^ y`**
- Se os atributos envolvidos forem dois objetos (**x** e **y**), que tenham boas implementações do **hashCode**, o **hashCode** da classe principal poderá ser obtido através de **`x.hashCode() ^ y.hashCode()`** (o uso do **`hashCode()`** reduz o problema à "combinação" de valores inteiros).

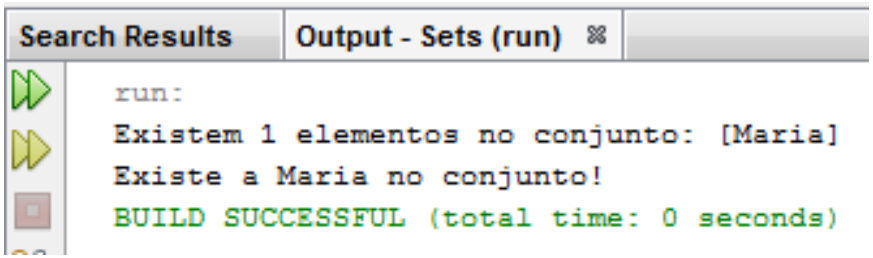
§- As considerações matemáticas e estatísticas subjacentes a estas escolhas saem do âmbito da cadeira de POO.

# Voltando ao exemplo de Set<Pessoa>

- Para o exemplo do conjunto de pessoas funcionar será, então, necessário, para além de redefinir o método **`equals`** (que apenas testava a igualdade do atributo **`nome`**), redefinir o método **`hashCode`**:

```
@Override
public int hashCode() {
  return nome.hashCode();
}
```

- A execução do programa já dará o resultado pretendido (não repete os elementos e consegue detetar a sua presença):



# Definição e utilização de classes

- Ao definirmos uma classe onde apenas alguns atributos identificam a unicidade do objeto (ex.: código, nomes que não podem ser repetidos, etc.) devemos definir:
  - Os seus atributos (como **`private`**);
  - Os construtores (validando os argumentos, gerando, eventualmente, exceções)
  - Os **`get`** e **`set`** (validando os argumentos, gerando, eventualmente exceções ou não permitindo as modificações) que façam sentido
  - O **`toString`** para obter uma representação legível dos objetos
  - Redefinir o **`equals`** e **`hashCode`** para se garantir a unicidade nas coleções (e noutras comparações)

# Resumindo

- **`Set`** – interface que representa um conjunto de elementos, onde não existem repetidos e onde a ordem é irrelevante
- **`HashSet`** – implementação baseada em **`hash table`**
- **`LinkedHashSet`** e **`TreeSet`** – implementações que garantem a ordem de iteração
- Utilizar **`For-Each`** ou **`iterator`** para percorrer os elementos
- Definir as variáveis baseadas na interface e não nas classes concretas
- Unicidade através do uso do método **`equals`**
- Uso do **`hashCode`** para otimizar as implementações
- Relação entre **`equals`** e **`hashCode`**
- Regras para redefinição do **`hashCode`**

# Leitura Complementar

- Capítulo 8
  - Páginas 253 a 344

- http://docs.oracle.com/javase/tutorial/collections/interfaces/index.html