Programação Orientada por Objetos

As coleções HashSet, HashMap

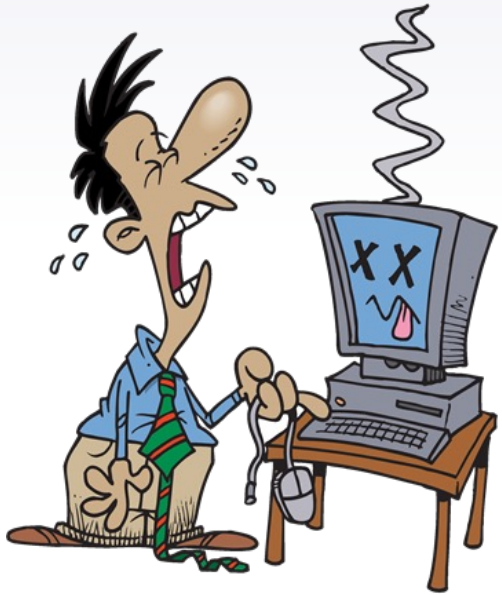
Prof. Cédric Grueau
Prof. José Sena Pereira
Departamento de Sistemas e Informática
Escola Superior de Tecnologia de Setúbal
Instituto Politécnico de Setúbal
2022/2023

Sumário

- Introdução às coleções
  - As coleções HashSet e HashMap.

Exemplo – Sistema de Apoio Técnico

- Sistema de apoio técnico:
  - Criado para substituir o apoio técnico dado aos clientes da empresa DodgySoft.
  - O antigo sistema permitia através do telefone, o esclarecimento de dúvidas e a resolução de problemas relacionados com os produtos da empresa.
  - Dadas as dificuldades pelas quais passa a empresa foi decidido acabar com o departamento de apoio técnico e construir um sistema que imitasse as respostas dos técnicos funcionando online e dando a sensação que o apoio técnico continuava a ser prestado.

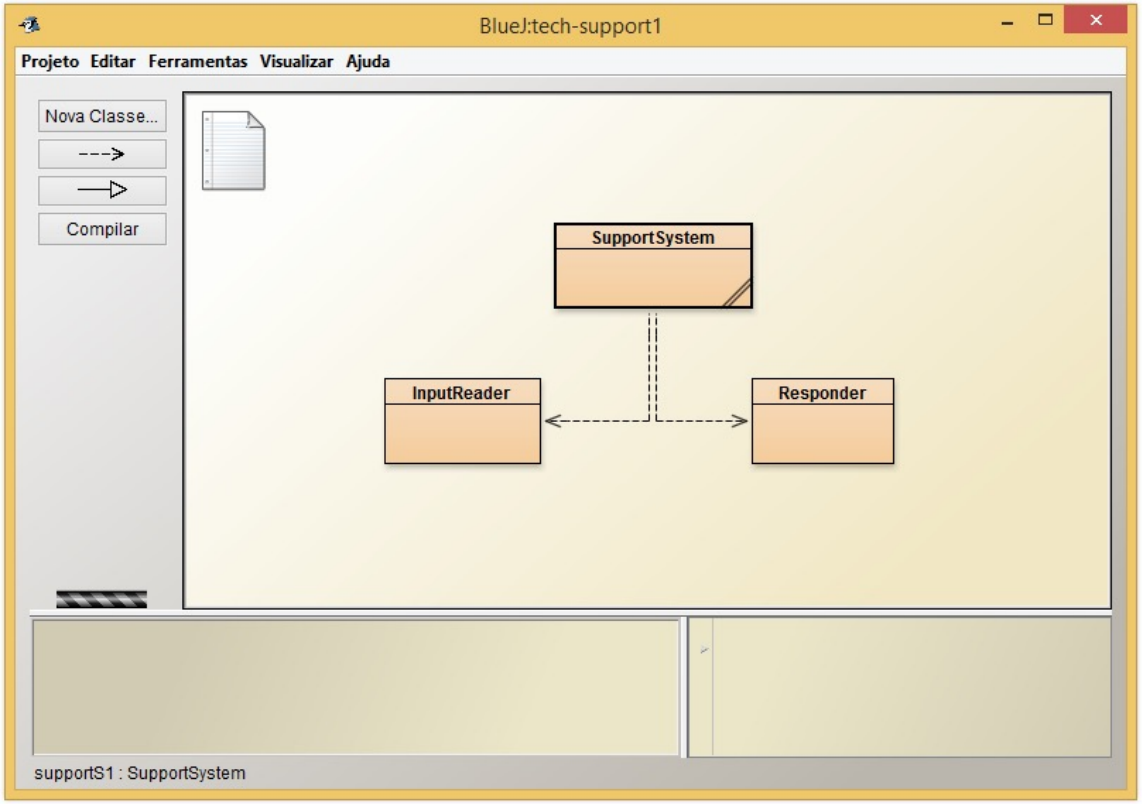


Exemplo – Sistema de Apoio Técnico

- Utilização (demonstração)

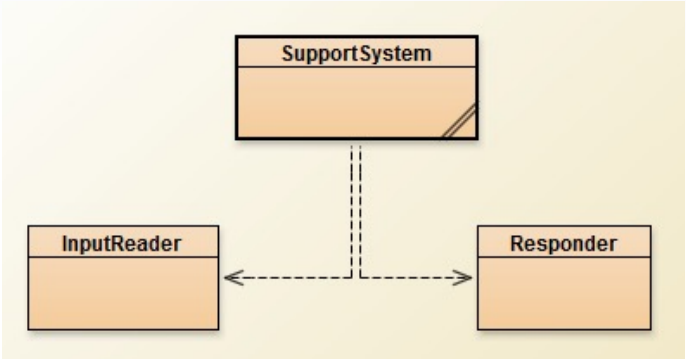
```
Welcome to the DodgySoft Technical Support System.

Please tell us about your problem.
We will assist you with any problem you might have.
Please type 'bye' to exit our system.
> help
That sounds interesting. Tell me more...
> My computer has a problem
That sounds interesting. Tell me more...
> nothing else
That sounds interesting. Tell me more...
> pleaseeeeeeeeeee
That sounds interesting. Tell me more...
> bye
Nice talking to you. Bye...
```

Exemplo – Sistema de Apoio Técnico



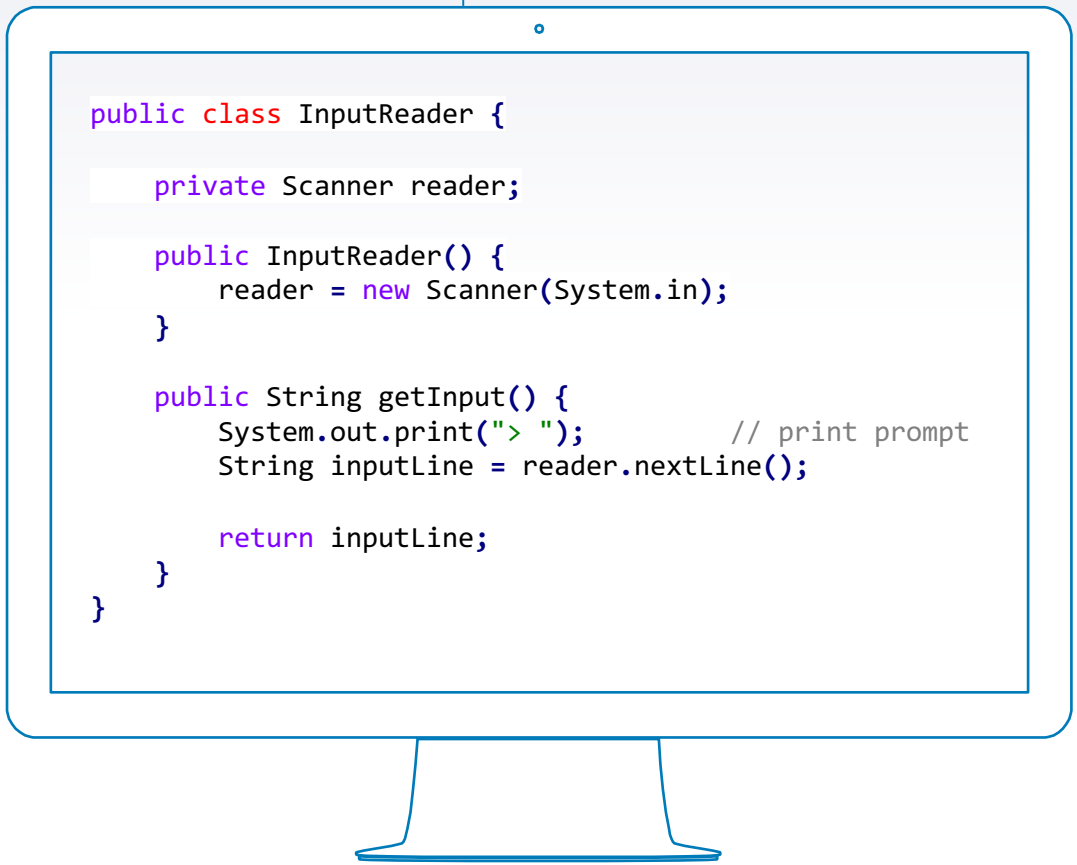
Exemplo – Sistema de Apoio Técnico

- Classes do tech support
  - InputReader – Usada para ler os dados do utilizador, representa o leitor de dados.
  - Responder – usada para gerar a resposta a ser dada ao utilizador.
  - SupportSystem – usada para gerir o sistema e interagir com o utilizador, lê o texto do utilizador e mostra a resposta.



Exemplo – Sistema de Apoio Técnico

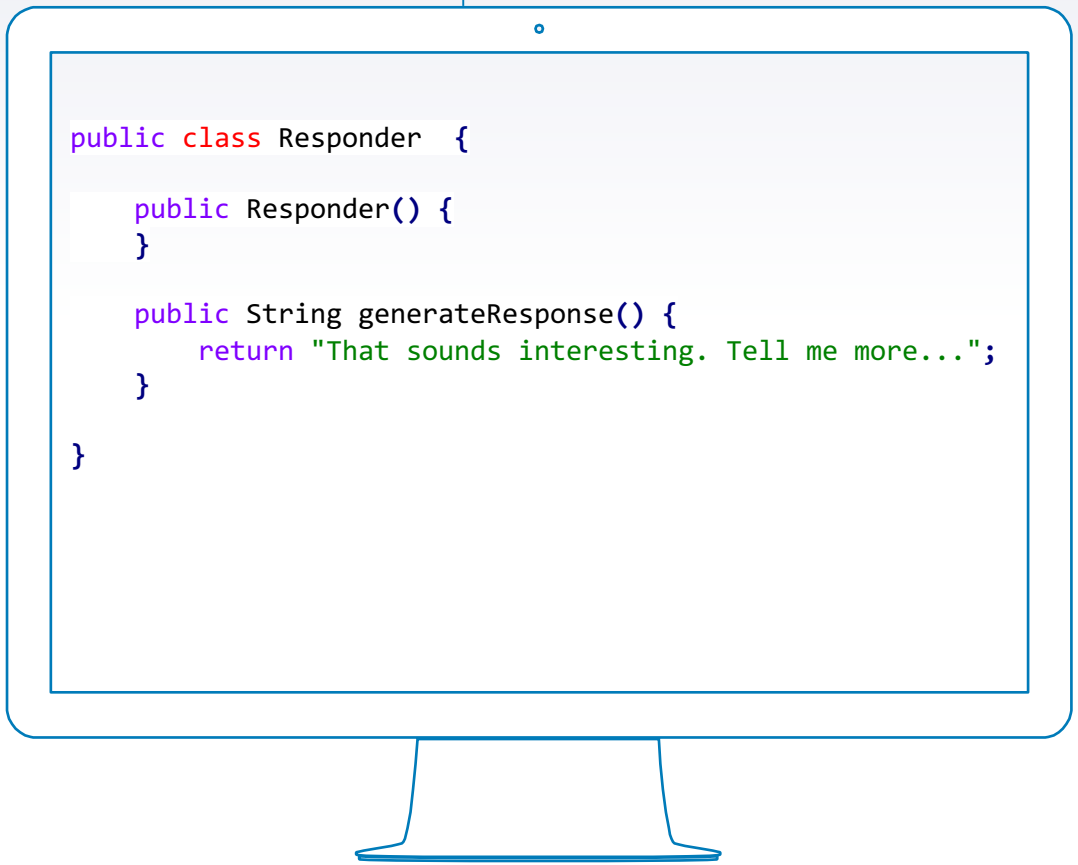
- Classe **InputReader**



```java
public class InputReader {

    private Scanner reader;

    public InputReader() {
        reader = new Scanner(System.in);
    }

    public String getInput() {
        System.out.print("> ");         // print prompt
        String inputLine = reader.nextLine();

        return inputLine;
    }
}
```

Exemplo – Sistema de Apoio Técnico

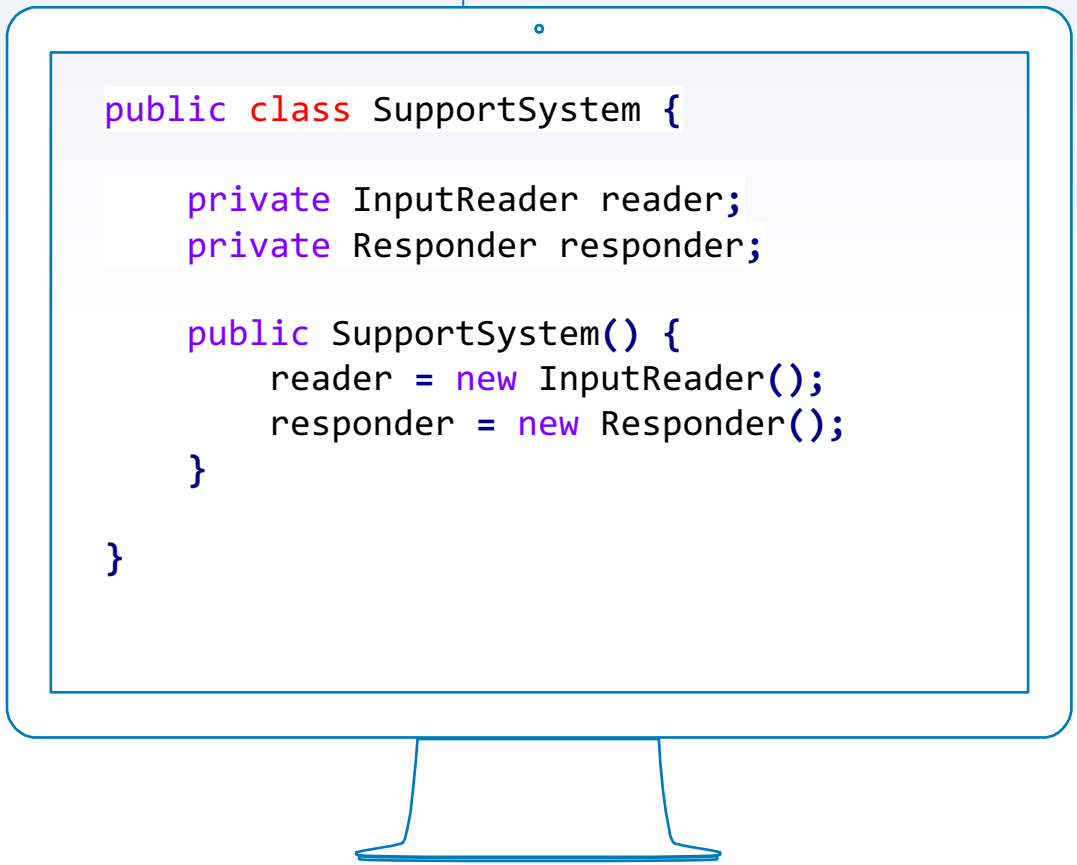
- Classe **Responder**



```
public class Responder  {

    public Responder() {
    }

    public String generateResponse() {
        return "That sounds interesting. Tell me more...";
    }

}
```

Exemplo – Sistema de Apoio Técnico

- Classe **SupportSystem**



```
public class SupportSystem {

    private InputReader reader;
    private Responder responder;

    public SupportSystem() {
        reader = new InputReader();
        responder = new Responder();
    }

}
```

Exemplo – Sistema de Apoio Técnico

- Classe **SupportSystem** – método **start**

```
public void start()  {
  boolean finished = false;

  printWelcome();

  while(!finished) {
     String input = reader.getInput();

    if(input.startsWith("bye")) {
       finished = true;
    }
    else {
      String response = responder.generateResponse();
      System.out.println(response);
        }
      }
        printGoodbye();
    }
```

Exemplo – Sistema de Apoio Técnico

- Classe **SupportSystem** – métodos **printWelcome** e **printGoodbye**

```
private void printWelcome() {
    System.out.println("Welcome to the DodgySoft Technical Support System.");
    System.out.println();
    System.out.println("Please tell us about your problem.");
    System.out.println("We will assist you with any problem you might
have.");
    System.out.println("Please type 'bye' to exit our system.");
}

private void printGoodbye() {
    System.out.println("Nice talking to you. Bye...");
}
```

Exemplo – Sistema de Apoio Técnico

- Sistema de Apoio Técnico:
  - Tal como está não faz grande coisa
    - A entrada de dados é sensível a espaços iniciais e finais e a caracteres maiúsculos e minúsculos.
    - A resposta é sempre igual
  - Melhoramentos iniciais
    - Ser mais flexível na entrada de dados
    - Ter mais respostas. Podemos selecionar aleatoriamente uma delas.

Exemplo – Sistema de Apoio Técnico (2)

- Classe **SupportSystem**
  – método **start** (versão 2)

Mais flexível na leitura dos dados.

```
public void start()  {
    boolean finished = false;
    printWelcome();
    while(!finished) {
        String input = reader.getInput().trim().toLowerCase();
        if(input.startsWith("bye")) {
            finished = true;
        }
        else {
            String response = responder.generateResponse();
            System.out.println(response);
        }
    }
    printGoodbye();
}
```

Exemplo – Sistema de Apoio Técnico (2)

```
public class Responder  {

    private Random randomGenerator;
    private ArrayList<String> responses;

    public Responder()  {
        randomGenerator = new Random();
        responses = new ArrayList<String>();
        fillResponses();
    }
    // Restante código
}
```

Para se escolher aleatoriamente uma resposta

Para guardar uma lista de respostas

Para criar a lista de respostas

- Classe **Responder** (versão 2)

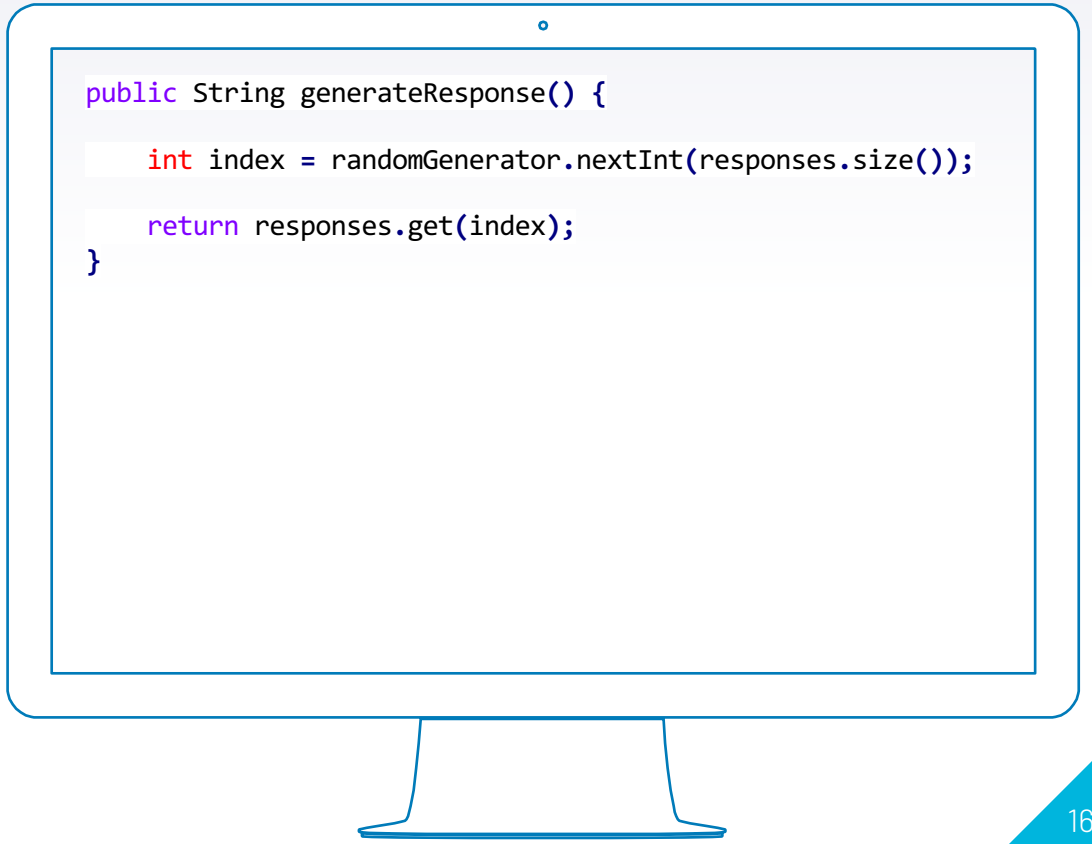
Exemplo – Sistema de Apoio Técnico (2)

- Classe **Responder** (2) – método **fillResponses**

```
private void fillResponses()  {

    responses.add("That sounds odd. Could you describe that problem in more detail?");
    responses.add("No other customer has ever complained about this before. \n" +
                  "What is your system configuration?");
    responses.add("That's a known problem with Vista. Windows 7 is much better.");
    responses.add("I need a bit more information on that.");
    responses.add("Have you checked that you do not have a dll conflict?");
    responses.add("That is explained in the manual. Have you read the manual?");
    responses.add("Your description is a bit wishy-washy. Have you got an expert\n" +
                  "there with you who could describe this more precisely?");
    responses.add("That's not a bug, it's a feature!");
    responses.add("Could you elaborate on that?");

}
```

Exemplo – Sistema de Apoio Técnico (2)

- Classe **Responder** (2) – método **generateResponse** (2)



```
public String generateResponse() {

    int index = randomGenerator.nextInt(responses.size());

    return responses.get(index);
}
```

Exemplo – Sistema de Apoio Técnico (2)

- Utilização (2)

```
Welcome to the DodgySoft Technical Support System.

Please tell us about your problem. We will assist you
with any problem you might have. Please type 'bye'
to exit our system.
> my computer doesn't start
I need a bit more information on that.
> nothing shown
Your description is a bit wishy-washy. Have you got an
expert
there with you who could describe this more precisely?
> yes
Could you elaborate on that?
> nothing done
Could you elaborate on that?
> bye
Nice talking to you. Bye...
```

Exemplo – Sistema de Apoio Técnico (2)

- Sistema de Apoio Técnico (versão 2):
  - Está melhor mas ainda tem alguns problemas:
    - As respostas não dependem do texto que o utilizador introduziu.
  - Melhoramentos finais
    - Ter várias respostas associando cada uma delas a uma palavra que possa existir no texto que o utilizador introduziu.
    - Ter uma coleção com as palavras que o utilizador introduziu.
  - Vamos usar uma nova classe de coleção – HashSet – para guardar as palavras do texto que o utilizador inseriu.
  - Vamos usar outra classe de coleção – HashMap – para guardar as ligações entre palavras e respostas associadas.

Classe HashSet

- A classe de coleção **HashSet**:
  - `import java.util.Hashset`
    - A classe de coleção HashSet representa um conjunto (set)
      - Nos conjuntos os elementos não se encontram ordenados.
        - Neste caso deixamos de saber a posição dos elementos porque simplesmente não se aplica (ao contrário das listas).
      - Nos conjuntos não existem elementos repetidos.
    - A classe HashSet é uma classe genérica
      - Tal como na classe ArrayList, recebe o tipo dos elementos como parâmetro:
        - Exemplo: `HashSet<Person>` , `HashSet<Student>`, etc.

Classe HashSet

- Métodos da classe HashSet:
  - Muitos dos métodos da classe HashSet são semelhantes aos usados pela classe `ArrayList` (e por outras classes de coleção):
  - `size` – saber o número de elementos que existem,
  - `isEmpty` – determinar se existem elementos,
  - `add` – adicionar um elemento,
    - Neste caso, se já existir, o elemento não é adicionado e retorna false
  - `remove` – remover um elemento,
    - Recebe como parâmetro o elemento a remover
  - `clear` – remover todos os elementos,
  - `contains` – verifica se um elemento existe na coleção.

Classe HashSet

- Métodos da classe HashSet:
  - Possui alguns métodos que permitem fazer as tradicionais operações matemáticas sobre conjuntos:
    - contains – operação de pertença $\in$,
    - addAll – operação de união $\cup$,
    - retainAll – operação de interseção $\cap$,
    - removeAll – operação de diferença –,
    - containsAll – operação de contenção $\subset$.

  Nota: os métodos acima que terminam em All recebem como parâmetro outra coleção.

Classe HashSet

- Classe **HashSet** - Exemplo

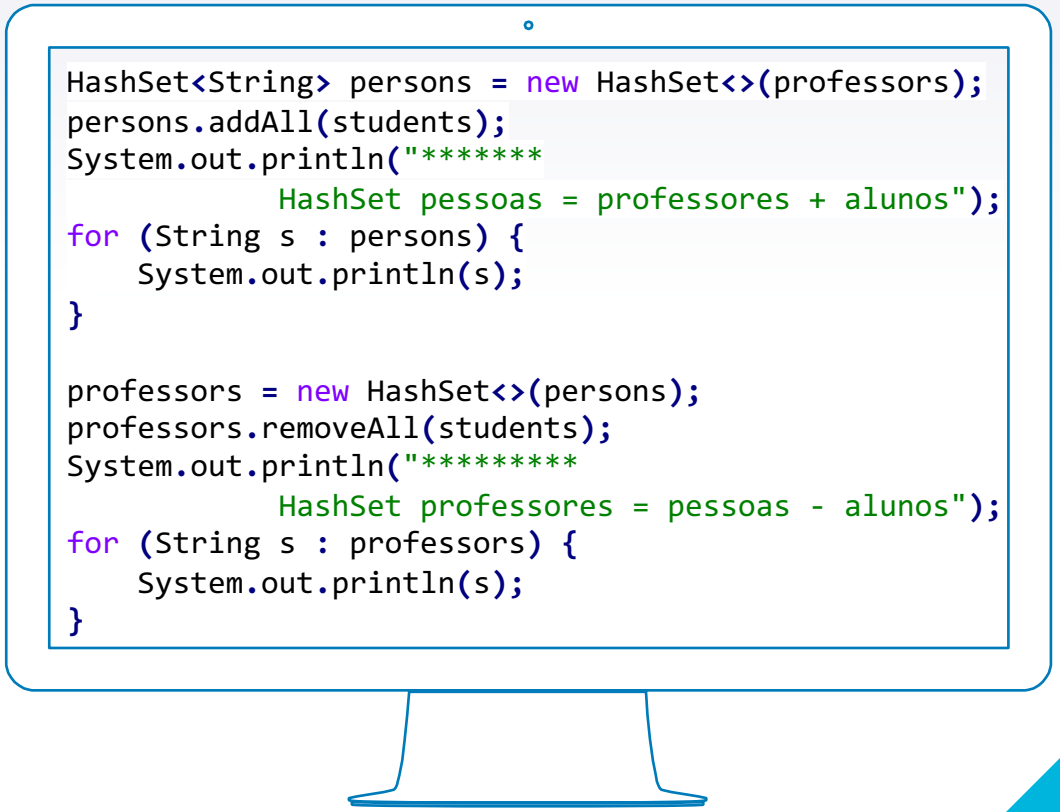
```
*** HashSet
professores
Joao
Ana
***** HashSet alunos
Joao
Luis
```

```
System.out.println("*** HashSet professores");
HashSet<String> professors = new HashSet<>();
professors.add("Ana");
professors.add("Joao");
for(String s: professors) {
    System.out.println(s);
}

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

Classe HashSet

- Classe **HashSet** - Exemplo

```
******* HashSet pessoas = professores + alunos
Joao
Ana
Luis
********* HashSet professores = pessoas - alunos
Ana
```



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

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

Classe HashSet

- Classe **HashSet** – Exemplo 2

```
true
false
[A, B, C]
[A, B]
[C]
true
```

```
HashSet<String> c1 = new HashSet<>();
c1.add("A");c1.add("B"); // c1 = { A, B }
HashSet<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"));
HashSet<String> union = new HashSet<>(c1);
union.addAll(c2);
System.out.println(union);
HashSet<String> intersection = new HashSet<>(c1);
intersection.retainAll(c2);
System.out.println(intersection);
HashSet<String> diference = new HashSet<>(c2);
diference.removeAll(c1);
System.out.println(diference);
System.out.println(c2.containsAll(c1));
```

Exemplo – Sistema de Apoio Técnico (3)

- Classe **InputReader** – método **getInput** (3)

```
public HashSet<String> getInput() {
    System.out.print("> ");                    // print prompt
    String inputLine = reader.nextLine().trim().toLowerCase();

    String[] wordArray = inputLine.split(" ");

    HashSet<String> words = new HashSet<String>();
    for(String word : wordArray) {
        words.add(word);
    }
    return words;
}
```

Divide o texto em palavras e retorna-as como um *array* (método da classe **String**)

Passa as palavras recebidas do array para um **Hashset** e assim elimina as repetições

Exemplo – Sistema de Apoio Técnico (3)

- Classe **SupportSystem** – método **start** (3)

```
boolean finished = false;
printWelcome();
while(!finished) {
    HashSet<String> input = reader.getInput();
    if(input.contains("bye")) {
        finished = true;
    }
    else {
       String response = responder.generateResponse(input);
       System.out.println(response);
        }
    }
    printGoodbye();
}
```

O *conjunto* de palavras que vêm no texto escrito pelo utilizador é passado para o método **generateResponse**

Classe HashMap

- A classe de coleção HashMap:
  - `import java.util.HashMap`
    - A classe de coleção **`HashMap`** representa um mapeamento (map)
      - Nos mapeamentos ou mapas são guardados pares de elementos.
      - Um dos elementos do par é a chave o outro é o valor. Dizemos que a cada chave está associado um valor.
        - As chaves são únicas, não podendo haver repetições.
        - Os valores podem ser repetidos desde que estejam associados a chaves diferentes.
    - A classe **`HashMap`** é uma classe genérica
      - No caso dos HashMap como temos pares de elementos devemos fornecer os tipos de cada um dos elementos do par como parâmetros:
        - Exemplo: `HashMap<Integer,Person>` , `HashMap<String,String>`, etc.

Classe HashMap

- A classe de coleção HashMap:
  - A classe de coleção HashMap é utilizada para a associação entre dois elementos, por exemplo:
    - Num dicionário temos palavras (chaves) associadas a definições (valores).
      - HashMap<String,String>
    - Numa lista telefónica podemos ter números de telefone associados a pessoas.
      - HashMap<Integer,Person>

Classe HashMap

- Métodos da classe HashMap:
  - Métodos comuns a outras coleções:
    - size – saber o número de elementos que existem,
    - isEmpty – determinar se existem elementos,
    - clear – remover todos os elementos.
  - Métodos comuns:
    - put – adicionar um par chave-valor,
      - Recebe a chave e o valor como parâmetros. Se a chave já existir substitui o valor que estava guardado pelo novo. Retorna o valor anterior ou null se a chave ainda não existia na coleção.
    - get – vai buscar um valor associado a uma chave,
      - Recebe como parâmetro a chave. Retorna o valor para essa chave ou null se a chave não existir.

Classe HashMap

- Métodos da classe HashMap:
  - Métodos comuns (continuação):
    - remove – remove da coleção um par chave-valor,
      - Recebe como parâmetro a chave. Retorna o valor associado à chave ou null se a chave não existir.
    - containsKey – verifica se já existe um elemento nas chaves,
    - containsValue – verifica se já existe um elemento nos valores,
    - keySet – retorna um conjunto com todas as chaves,
    - values – retorna uma coleção com todos os valores,
    - entrySet – retorna um conjunto de objetos Map.Entry<K,V>.
      - Cada elemento do conjunto retornado tem a chave e o valor e é possível obter esses elementos usando, respetivamente, os métodos getKey() e getValue()

Classe HashMap

- Classe **HashMap** – Exemplo

```
Pessoas:
37 - Marco
23 - Maria
43 - Manuel Matos
13 - Maria
```

Estamos a aceder aos elementos através do conjunto das chaves

```
HashMap<Integer, String> mapNames = new HashMap<>();
mapNames.put(13, "Maria");
mapNames.put(43, "Manuel");
mapNames.put(37, "Marco");
mapNames.put(23, "Maria");         //Valor repetido
mapNames.put(43, "Manuel Matos"); //Chave repetida

System.out.println("Pessoas:");
for (Integer i : mapNames.keySet()) {
    System.out.println(i + " - " + mapNames.get(i));
}
```

Classe HashMap

- Classe **HashMap** – Exemplo 2

```
Pessoas:
Marco
Maria
Manuel Matos
Maria
```

Estamos a aceder apenas à coleção dos valores (nomes)

```
HashMap<Integer, String> mapNames = new HashMap<>();
mapNames.put(13, "Maria");
mapNames.put(43, "Manuel");
mapNames.put(37, "Marco");
mapNames.put(23, "Maria");          //Valor repetido
mapNames.put(43, "Manuel Matos"); //Chave repetida

System.out.println("Pessoas:");
for (String name : mapNames.values()) {
    System.out.println(name);
}
```

Classe HashMap

- Classe **HashMap** – Exemplo 3

```
Pessoas:
37 - Marco
23 - Maria
43 - Manuel Matos
13 - Maria
```

```
HashMap<Integer, String> mapNames = new HashMap<>();
mapNames.put(13, "Maria");
mapNames.put(43, "Manuel");
mapNames.put(37, "Marco");
mapNames.put(23, "Maria");          //Valor repetido
mapNames.put(43, "Manuel Matos"); //Chave repetida
System.out.println("Pessoas:");
for (Map.Entry pair : mapNames.entrySet()) {
    System.out.println(pair.getKey() + " - "
        + pair.getValue());
}
```

Estamos a aceder ao conjunto das *entradas* do mapa

Exemplo – Sistema de Apoio Técnico (3)

- Classe **Responder** (3)

```
public class Responder {
    // Usado para associar palavras a respostas.
    private HashMap<String, String> responseMap;
    // Lista de respostas se não existirem palavras reconhecidas.
    private ArrayList<String> defaultResponses;
    private Random randomGenerator;

    public Responder() {
        responseMap = new HashMap<String, String>();
        defaultResponses = new ArrayList<String>();
        fillResponseMap();
        fillDefaultResponses();
        randomGenerator = new Random();
    }
    // restantes métodos
```

Exemplo – Sistema de Apoio Técnico (3)

- Classe **Responder** (3) – métodos **fillDefaultResponses** e **pickDefaultResponse**

```
private void fillDefaultResponses() {
    defaultResponses.add("That sounds odd. Could you describe that problem in more" +
                         "detail?");
    defaultResponses.add("No other customer has ever complained about this before. \n" +
                         "What is your system configuration?");
    defaultResponses.add("That sounds interesting. Tell me more...");
    defaultResponses.add("I need a bit more information on that.");
    defaultResponses.add("Have you checked that you do not have a dll conflict?");
    defaultResponses.add("That is explained in the manual. Have you read the manual?");
    defaultResponses.add("Your description is a bit wishy-washy. Have you got an expert\n“
                       + "there with you who could describe this more precisely?");
    defaultResponses.add("That's not a bug, it's a feature!");
    defaultResponses.add("Could you elaborate on that?");
}
private String pickDefaultResponse() {
    int index = randomGenerator.nextInt(defaultResponses.size());
    return defaultResponses.get(index);
}
```

Exemplo – Sistema de Apoio Técnico (3)

- Classe **Responder** (3) – método **fillResponsesMap**

```
private void fillResponseMap() {
    responseMap.put("crash",
                    "Well, it never crashes on our system. It must have something\n" +
                    "to do with your system. Tell me more about your configuration.");
    responseMap.put("crashes",
                    "Well, it never crashes on our system. It must have something\n" +
                    "to do with your system. Tell me more about your configuration.");
    responseMap.put("slow",
                    "I think this has to do with your hardware. Upgrading your processor\n"
                  + "should solve all performance problems. Have you got a problem with\n"
                  + "our software?");
    responseMap.put("windows",
                "This is a known bug to do with the Windows operating system. Please\n" +
                "report it to Microsoft. There is nothing we can do about this.");
    responseMap.put("bug",
                "Well, you know, all software has some bugs. But our software engineers\n"
            +   "are working very hard to fix them. Can you describe the problem a bit\n" +
                "further?");

    // outras associações omitidas

}
```

Exemplo – Sistema de Apoio Técnico (3)

- Classe **Responder** (3) – método **generateResponse**

```
public String generateResponse(HashSet<String> words) {
    for (String word : words) {
        String response = responseMap.get(word);
        if(response != null) {
            return response;
        }
    }

    return pickDefaultResponse();
}
```

Exemplo – Sistema de Apoio Técnico (3)

- Utilização (3)

```
Welcome to the DodgySoft Technical Support System.

Please tell us about your problem.
We will assist you with any problem you might have.
Please type 'bye' to exit our system.
> i have a hardware problem
No other customer has ever complained about this before.
What is your system configuration?
> a windows computer
This is a known bug to do with the Windows operating system.
Please
report it to Microsoft. There is nothing we can do about
this.
> bye
Nice talking to you. Bye...
```

Exemplo – Coleção de Cromos

- Requisitos da aplicação:
  - Fazer a gestão de uma coleção de cromos.
  - Cada cromo é caracterizado pelo seu número e pelo seu estado (bom, razoável, mau).
  - Na gestão dos cromos deverá ser definido o nome da coleção e o número de cromos da coleção completa. Os melhores cromos devem ser guardados numa coleção sem repetições para irem para a caderneta. Os repetidos serão guardados separadamente.
  - Na gestão precisamos saber quantos cromos temos, quantos faltam, adicionar cromos, lista de repetidos, lista dos que temos (números).

Exemplo – Coleção de Cromos

- Classe **TradingCard**

```
public enum CardState {
     GOOD, REASONABLE, POOR
}

public class TradingCard {

     private int number;
     private CardState state;

     public TradingCard(int number, CardState state) {
          this.number = number;
          this.state = state;
     }

     // restante código omitido
}
```

Exemplo – Coleção de Cromos

- Classe **TradingCard**

```
public class TradingCard {
    // restante código omitido

    public CardState getState() {
        return state;
    }

    public void setState(CardState state) {
        this.state = state;
    }

    public int getNumber() {
        return number;
    }

    public String toString() {
        return "#" + number + " (" + state + ")";
    }
}
```

Exemplo – Coleção de Cromos

```java
public class TradingCardCollection {

    private String title;
    private int largestNumber;
    private ArrayList<TradingCard> repeated;
    private HashSet<TradingCard> cards;

    public TradingCardCollection(String title, int largestNumber) {

        this.title = title;
        this.largestNumber = largestNumber;
        repeated = new ArrayList<>();
        cards = new HashSet<>();
    }

    // restante código omitido
}
```

Lista para os repetidos

Conjunto para os cromos da caderneta

- Classe **TradingCardCollection**

Exemplo – Coleção de Cromos

- Classe **TradingCardCollection** – **validateTradingCard** e **addTradingCard**

```
private boolean validateTradingCard(TradingCard tradingCard) {
    return tradingCard != null &&
           tradingCard.getNumber() > 0 &&
           tradingCard.getNumber() <= largestNumber ;
}


public void addTradingCard(TradingCard tradingCard) {
    if (validateTradingCard(tradingCard)) {
        if (!cards.add(tradingCard)) {
            repeated.add(tradingCard);
        }
    }
}
```

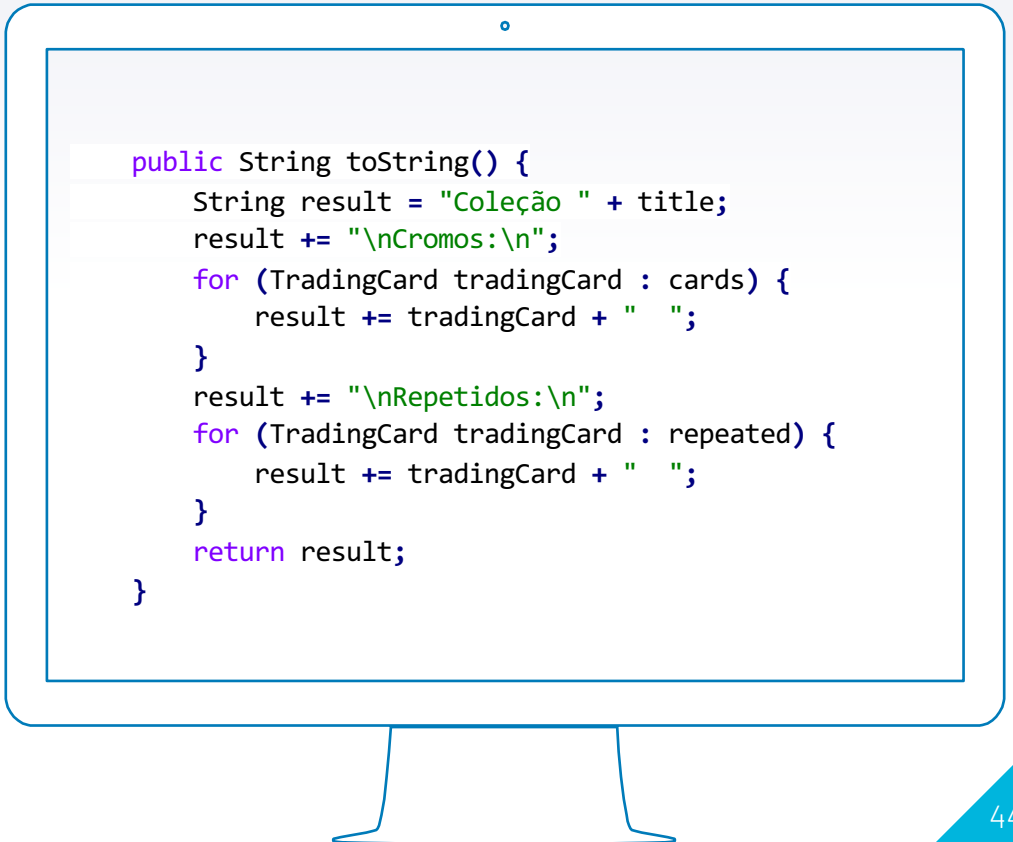
Se já existir na coleção adicionamos aos repetidos

Exemplo – Coleção de Cromos

- Classe

  TradingCardCollection –

  toString



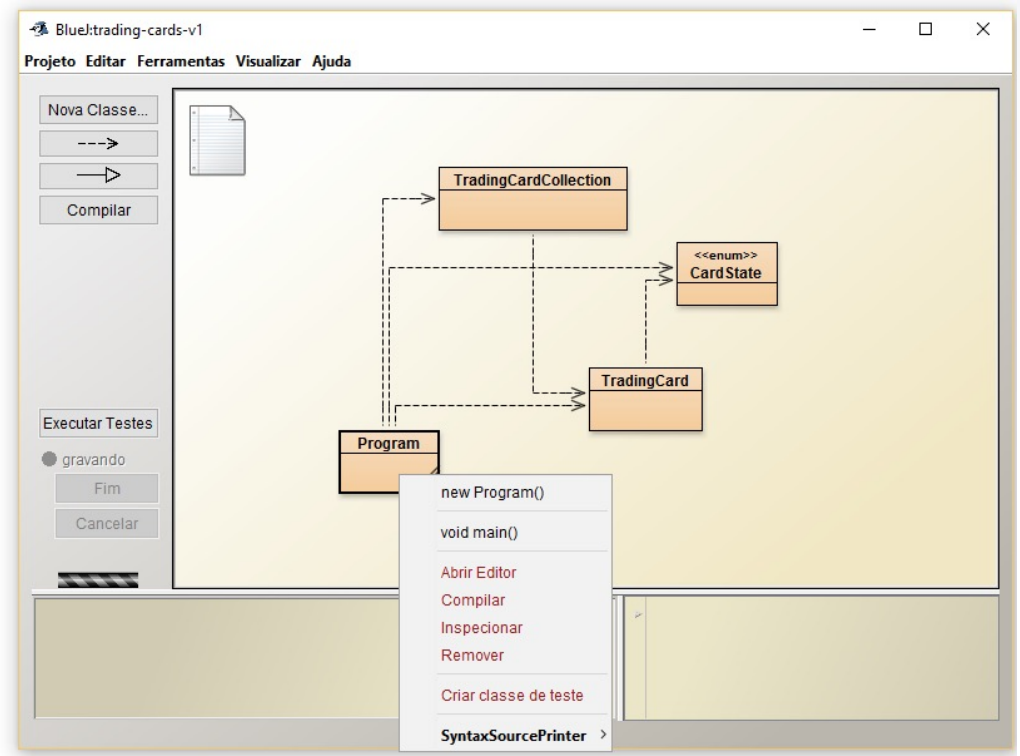
```
public String toString() {
    String result = "Coleção " + title;
    result += "\nCromos:\n";
    for (TradingCard tradingCard : cards) {
        result += tradingCard + "  ";
    }
    result += "\nRepetidos:\n";
    for (TradingCard tradingCard : repeated) {
        result += tradingCard + "  ";
    }
    return result;
}
```

Exemplo – Coleção de Cromos

- Teste simples da aplicação

```
public class Program {

    public static void main() {
        TradingCard card1 = new TradingCard(12, CardState.GOOD);
        TradingCard card2 = new TradingCard(12, CardState.GOOD);
        TradingCard card3 = new TradingCard(5, CardState.POOR);
        TradingCard card4 = new TradingCard(6, CardState.GOOD);

        TradingCardCollection cards = new TradingCardCollection("Teste", 20);
        cards.addTradingCard(card1);
        cards.addTradingCard(card2);
        cards.addTradingCard(card3);
        cards.addTradingCard(card4);

        System.out.println(cards);
    }
}
```

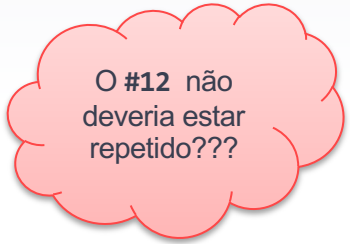
Exemplo – Coleção de Cromos



Exemplo – Coleção de Cromos

Teste simples da aplicação

```
TradingCard card1 = new TradingCard(12, CardState.GOOD);
TradingCard card2 = new TradingCard(12, CardState.GOOD);
TradingCard card3 = new TradingCard(5, CardState.POOR);
TradingCard card4 = new TradingCard(6, CardState.GOOD);

TradingCardCollection cards = new TradingCardCollection("Teste", 20);
cards.addTradingCard(card1);
cards.addTradingCard(card2);
cards.addTradingCard(card3);
cards.addTradingCard(card4);

System.out.println(cards);
```




```
Coleção Teste
Cromos:
#6 (GOOD)  #12 (GOOD)  #5 (POOR)  #12 (GOOD)
Repetidos:
```

Classe HashSet

- A classe HashSet usa um método chamado equals para comparar dois elementos.
  - O método `equals`, tal como o método toString existe em todos os objetos e é usado para comparar dois objetos. Por omissão, compara as referências dos objetos.
  - A assinatura deste método é:

```
public boolean equals(Object obj)
```

    - O objeto recebido (tipo Object) pode ser de qualquer classe mas espera-se que seja da classe do objeto a comparar porque senão não será igual (segundo if do slide seguinte).

Classe HashSet

- Modo de criação do método **`equals`** (para qualquer classe)
  - Modelo:

```
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final ClasseDoObjeto other = (ClasseDoObjeto) obj;

    return //teste da igualdade entre os dois objetos (this e other;
}
```

O `getClass` à semelhança do `toString`, `equals` e `hashCode` é outro dos métodos que todas as classes têm por omissão

Classe Cromo

- O método **equals** que queremos que seja utilizado na situação atual, deverá ser definido na classe **TradingCard** e será (apenas compara o números dos cromos):

```
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final TradingCard other = (TradingCard) obj;

    return this.number == other.number;
}
```

Classe HashSet

- A classe `HashSet` usa o método `equals` para comparar dois elementos mas não é ainda suficiente para que a comparação é seja feita eficientemente para todos os elementos da coleção. Para que isso aconteça é necessário fornecer mais um método: o método `hashCode`
  - O método hashCode, tal como os métodos toString e equals existe também em todos os objetos e tem uma implementação por omissão que será necessária alterar na maioria dos casos.
  - A assinatura deste método é:

    ```
    public int hashCode()
    ```

    - O valor inteiro retornado deverá ser diferente sempre que os objetos a comparar forem diferentes.

Classe HashSet – 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 boolean add (Object object) {
    int position = object.hashCode();
    if (values[position] == null) { //ainda não foi colocado
        values[position] = object;
        return true;
    }
    return false;
}
```

- 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

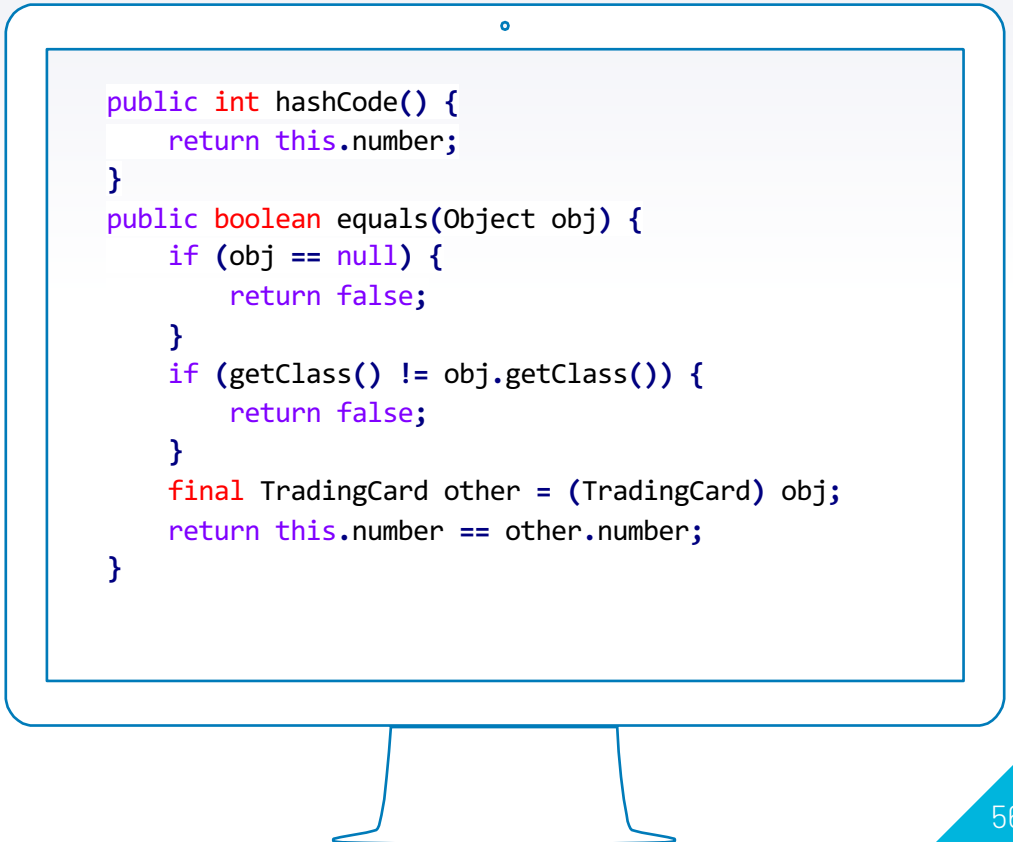
- Ao se introduzir um elemento num **`HashSet`** o sistema começa por determinar a posição do elemento na hash table através da chamada ao método **`hashCode`**, percorrendo em seguida a sequência de elementos comparando-os através do método **`equals`**.
- Por esta razão os métodos **`equals`** e **`hashCode`** 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.

Definição do método hashCode

- Na definição do método hashCode do Java, dada em http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html é dito que:
  - Durante a uma execução de uma aplicação Java, a chamada ao método hashCode, para o mesmo objeto, deve devolver valores iguais. Tal não é garantido para execuções 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.

Exemplo – Coleção de Cromos

- Classe **TradingCard** – **equals** e **hashCode**



```
public int hashCode() {
    return this.number;
}
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final TradingCard other = (TradingCard) obj;
    return this.number == other.number;
}
```

Exemplo – Coleção de Cromos

- Teste simples da aplicação

```
Coleção Teste
Cromos:
#5 (POOR)  #6 (GOOD)  #12 (GOOD)
Repetidos:
#12 (GOOD)  #12 (REASONABLE)
```

```
TradingCard card1 = new TradingCard(12, CardState.GOOD);
TradingCard card2 = new TradingCard(12, CardState.GOOD);
TradingCard card3 = new TradingCard(5, CardState.POOR);
TradingCard card4 = new TradingCard(6, CardState.GOOD);
TradingCard card5 = new TradingCard(12, CardState.REASONABLE);

TradingCardCollection cards = new TradingCardCollection("Teste", 20);
cards.addTradingCard(card1);
cards.addTradingCard(card2);
cards.addTradingCard(card3);
cards.addTradingCard(card4);
cards.addTradingCard(card5);

System.out.println(cards);
```

Exemplo – Coleção de Cromos

- Classe **TradingCardCollection** – **isFull**, **isEmpty** e **totalCards**

```
public boolean isFull() {
    return cards.size() == largestNumber;
}
public boolean isEmpty() {
    return cards.isEmpty();
}
public int totalCards() {
    return cards.size() + repeated.size();
}
```

Exemplo – Coleção de Cromos

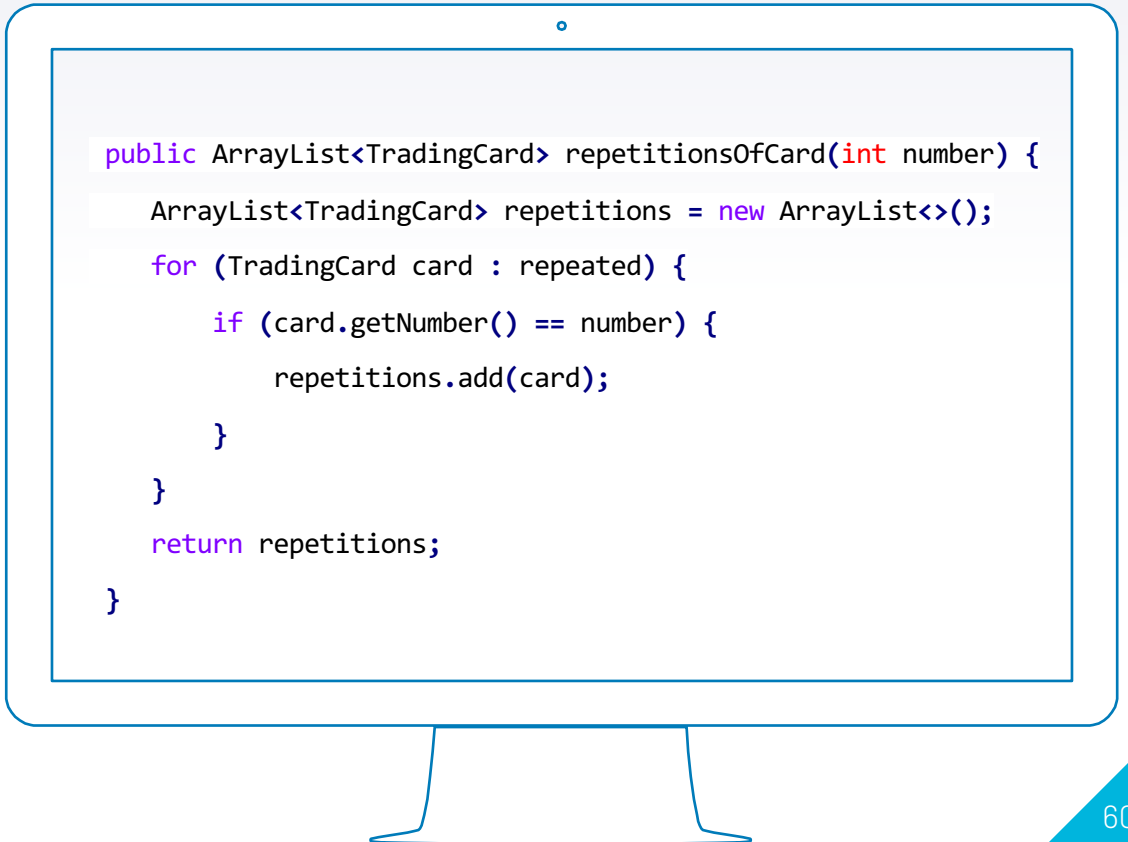
Classe **TradingCardCollection** –

addTradingCards e **missingCards**

```
public void addTradingCards(TradingCard[] cards) {
     for (TradingCard card : cards) {
        addTradingCard(card);
     }
 }
public HashSet<Integer> missingCards() {
   HashSet<Integer> missing = new HashSet<>();
     for (int i = 1; i <= largestNumber; i++) {
        missing.add(i);
     }
     for (TradingCard card : cards) {
        missing.remove(new Integer(card.getNumber()));
     }
     return missing;
}
```

Exemplo – Coleção de Cromos

- Classe **TradingCardCollection** – **repetitionsOfCard**



```
public ArrayList<TradingCard> repetitionsOfCard(int number) {
    ArrayList<TradingCard> repetitions = new ArrayList<>();
    for (TradingCard card : repeated) {
        if (card.getNumber() == number) {
            repetitions.add(card);
        }
    }
    return repetitions;
}
```

Exemplo – Coleção de Cromos

```
public void optimize() {
    HashSet<TradingCard> notGood = new HashSet<>();
    for (TradingCard card : cards) {
        if (card.getState() != CardState.GOOD) {
            notGood.add(card);
        }
    }
    for (TradingCard card : notGood) {
        for(int i=0; i<repeated.size(); i++){
            TradingCard iCard = repeated.get(i);
            if (iCard.equals(card) && iCard.isInBetterStateThan(card)) {
                cards.remove(card);
                cards.add(iCard);
                repeated.set(i, card);
                break;
            }
        }
    }
}
```

- Classe **TradingCardCollection** – **optimize**

Novo método:
`isInBetterStateThan`

Exemplo – Coleção de Cromos

- Classe **TradingCard** –

isInBetterStateThan

```
public boolean isInBetterStateThan(TradingCard card) {

     return (this.state == CardState.GOOD &&
               card.getState() != CardState.GOOD)  ||

              (this.state == CardState.REASONABLE &&
               card.getState() == CardState.POOR);
 }
```

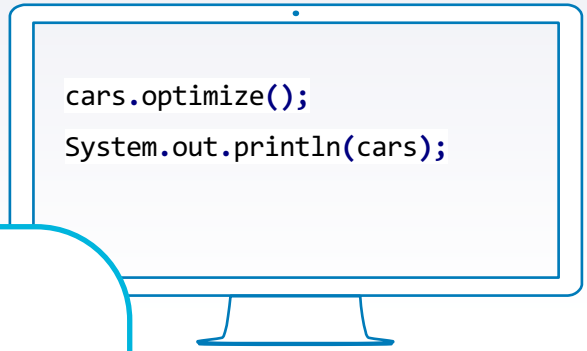
Exemplo – Coleção de Cromos

- Teste simples da aplicação

```
TradingCard[] cards = new TradingCard[10];
cards[0] = new TradingCard(1, CardState.GOOD);
cards[1] = new TradingCard(2, CardState.GOOD);
cards[2] = new TradingCard(2, CardState.REASONABLE);
cards[3] = new TradingCard(3, CardState.POOR);
cards[4] = new TradingCard(4, CardState.GOOD);
cards[5] = new TradingCard(7, CardState.GOOD);
cards[6] = new TradingCard(8, CardState.REASONABLE);
cards[7] = new TradingCard(8, CardState.GOOD);
cards[8] = new TradingCard(8, CardState.REASONABLE);
cards[9] = new TradingCard(8, CardState.POOR);

TradingCardCollection cars = new TradingCardCollection("Carros 2015", 10);

cars.addTradingCards(cards);
cars.addTradingCard(new TradingCard(3, CardState.REASONABLE));

System.out.println(cars);

System.out.println("Faltam: ");
for(int number : cars.missingCards())
    System.out.print(number + ", ");
System.out.println();
```

Exemplo – Coleção de Cromos

- Teste simples da aplicação




```
Coleção Carros 2015
Cromos:
#1 (GOOD)  #2 (GOOD)  #3 (POOR)  #4 (GOOD)  #7 (GOOD)  #8 (REASONABLE)
Repetidos:
#2 (REASONABLE)  #8 (GOOD)  #8 (REASONABLE)  #8 (POOR)  #3 (REASONABLE)
Faltam:
5, 6, 9, 10,
Coleção Carros 2015
Cromos:
#1 (GOOD)  #2 (GOOD)  #3 (REASONABLE)  #4 (GOOD)  #7 (GOOD)  #8 (GOOD)
Repetidos:
#2 (REASONABLE)  #8 (REASONABLE)  #8 (POOR)  #3 (POOR)  #8 (REASONABLE)
```

Exemplo – Coleção de Cromos (2)

```
public class TradingCardCollection {

    private String title;
    private int largestNumber;
    private ArrayList<TradingCard> repeated;
    private HashMap<Integer, TradingCard> cards;

    public TradingCardCollection(String title, int largestNumber) {
        this.title = title;
        this.largestNumber = largestNumber;
        repeated = new ArrayList<>();
        cards = new HashMap<>();
    }

    // restante código omitido
}
```

Lista para os repetidos

Mapa para os cromos da caderneta

- Classe **TradingCardCollection** com **HashMap**

Exemplo – Coleção de Cromos

- Classe **TradingCardCollection** com **HashMap** – **validateTradingCard** e **addTradingCard**

```
private boolean validateTradingCard(TradingCard tradingCard) {
    return tradingCard != null &&
           tradingCard.getNumber() > 0 &&
           tradingCard.getNumber() <= largestNumber ;
}

public void addTradingCard(TradingCard tradingCard) {
    if (validateTradingCard(tradingCard)) {
        if (!cards.containsKey(tradingCard.getNumber())) {
            cards.put(tradingCard.getNumber(), tradingCard);
        } else {
            repeated.add(tradingCard);
        }
    }
}
```

Se já existir na coleção adicionamos aos repetidos

Exemplo – Coleção de Cromos (2)

- Classe **TradingCardCollection**

  com **HashMap** – **missingCards**

```
public HashSet<Integer> missingCards() {
   HashSet<Integer> missing = new HashSet<>();
   for (int i = 1; i <= largestNumber; i++) {
       missing.add(i);
    }
    missing.removeAll(cards.keySet());
    return missing;
    }
```

Bibliografia

- Objects First with Java (6th Edition), David Barnes & Michael Kölling, Pearson Education Limited, 2016
  - Capítulo 6 (6.1 a 6.9)