Altar Code Labs

Advanced Features

Java SE 4/4

UP ACADEMY
LEARNING FOR SUCCESS

CONTENT

1. Java New Features
    - Enums
    - Annotations
    - Lambda expressions
    - Streams
2. Collections
    - Generics
    - Data Structures

Java New Features

- *Enums*
- *Annotations*
- *Lambda expressions*
- *Streams*

Enums

- Um enum é um tipo especial de dados que permite definir previamente um conjunto limitado instâncias

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
```

- Tem algumas características das classes:
  - Atributos
  - Métodos
  - Possibilidade de implementar interfaces

Enums

- Tem algumas limitações:
  - Não pode estender outros elementos pois implicitamente todos os enums estendem a classe Enum
  - Apenas pode ter construtores privados
  - Não permite um número variável de instâncias

- Disponibiliza alguns métodos estáticos genéricos:
  - **valueOf**(String): devolve a instância do enum com o nome fornecido
  - **valueOf**(Class<T>, String): devolve a instância do enum com a classe e o nome fornecidos
  - **values**(): devolve um array com todas as instâncias do enum

Exemplo de enum

```
public enum TesteEnum {
    SUNDAY("Dom", 1), MONDAY("Seg", 2), TUESDAY("Ter", 3), WEDNESDAY("Qua", 4),
    THURSDAY("Qui", 5), FRIDAY("Sex", 6), SATURDAY("Sab", 7);

    private String name;
    private int number;

    private TesteEnum(String name, int number) {
        this.name = name;
        this.number = number;
    }

    public String getName() {
        return name;
    }

    public int getNumber() {
        return number;
    }
}
```

```
public String traducao(TesteEnum teste) {
    // Podia ser feito no switch
    // So esta aqui para efeitos de demo
    if(teste == TesteEnum.MONDAY) {
        mondays++;
    }

    switch (teste) {
        case MONDAY:
        case TUESDAY:
        case WEDNESDAY:
        case THURSDAY:
        case FRIDAY:
            weekDays++;
            break;
        case SATURDAY:
        case SUNDAY:
            weekendDays++;
            break;
        default:
            System.out.println("Erro???");
    }

    return teste.toString();
}
```

Anotações

- Uma anotação é uma forma de metadados
- Não têm efeito direto na execução do código que anotam
- Pode ter ou não parâmetros e, quando existem, podem ser opcionais
- Qualquer projeto pode criar as suas próprias anotações
- Algumas utilizações das anotações:
- Informação para o compilador: detetar erros ou suprimir warnings
- Processamento em compile-time e deployment-time: geração de código, configuração, etc.
- Processamento em runtime: o programa realiza ações baseado nas anotações existentes

Aplicar anotações

- Em classes
- Em métodos
- Atributos de uma classe
- Variáveis locais
- Anotações
- E outras... (consultar o enumerado: java.lang.annotation.ElementType)

```
 */
@Entity
public class Post extends Abst
```

```
@PrePersist
public void setCurrentDate() {
    postDateTime = new Date();
}
```

```
@Lob
private String text;
```

```
@SuppressWarnings("rawtypes")
List ola = new ArrayList();
```

```
@Target({TYPE, FIELD, METHOD, PARAME
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings {
```

Criar anotações

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.TYPE})
public @interface Logged {

}
```

```
public @interface MyAnnotation {
    String name() default "";
    int numberOfTries() default 5;
    long[] historyTriesPerMonth() default {};
    int value();
}
```

```
@MyAnnotation(5)
public static void toPrintValues(Path file1) {
    System.out.println(file1.getFileName());
```

```
@MyAnnotation(value = 5, name = "LMR", historyTriesPerMonth = {6, 8, 10})
public static void toPrintValues(Path file1) {
    System.out.println(file1.getFileName());
```

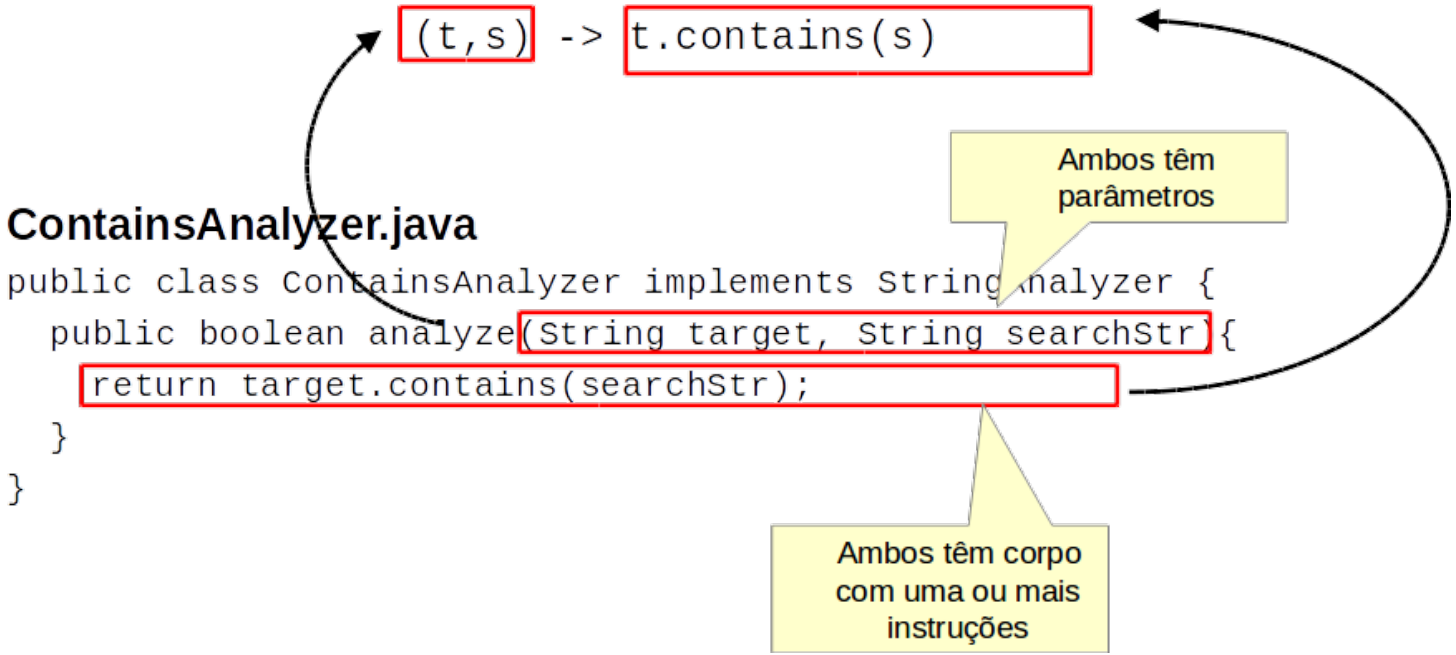
Expressões Lambda

- Funcionalidade introduzida no Java 8 que consiste em permitir passar um método como parâmetro para um outro método
- O método destino deve declarar como parâmetro uma interface funcional (interface apenas com um método abstrato)
- Exitem já diversas interfaces funcionais no package java.util.function

Lista de Parâmetros	Arrow Token	Corpo
`(int x, int y)`	`->`	`x + y`

- Exemplos:
  - (int x, int y) -> x + y
  - (x, y) -> x + y
  - (x, y) -> { system.out.println(x + y);}
  - (String s) -> s.contains("word")

Expressões Lambda



Expressões Lambda

- Expressões lambda com sintaxe mais curta

```
20      // Use short form Lambda
21      System.out.println("==Contains==");
22      Z06Analyzer.searchArr(strList01, searchStr,
23        (t, s) -> t.contains(s));
24
25      // Changing logic becomes easy
26      System.out.println("==Starts With==");
27      Z06Analyzer.searchArr(strList01, searchStr,
28        (t, s) -> t.startsWith(s));
```

- Os argumentos do método `searchArr` são:

```
public static void searchArr(String[] strList, String
  searchStr, StringAnalyzer analyzer)
```

Expressões Lambda

- As expressões lambda podem ser tratadas como variáveis
- Podem ser atribuidas, passadas como parâmetros e reutilisadas

```
19      // Lambda expressions can be treated like variables
20      StringAnalyzer contains = (t, s) -> t.contains(s);
21      StringAnalyzer startsWith = (t, s) -> t.startsWith(s);
22
23      System.out.println("==Contains==");
24      Z07Analyzer.searchArr(strList, searchStr,
25        contains);
26
27      System.out.println("==Starts With==");
28      Z07Analyzer.searchArr(strList, searchStr,
29        startsWith);
```

Interfaces funcionais

- O programador pode sempre criar as suas interfaces funcionais (apenas com um método abstrato)
- É aconselhável (embora não seja obrigatório) anotá-las com @FunctionalInterface
- Algumas interfaces funcionais já fornecidas pelo Java:
  - **Predicate**: Uma expressão que devolve um booleano
  - **Consumer**: Uma expressão que realiza operações sobre um objeto passado como argumento e que retorna void
  - **Function**: Transforma um T num U
  - **Supplier**: Fornece uma instância de T (como uma factory)
  - Variações primitivas das interfaces anteriores
  - Variações binárias (dois parâmetros) das interfaces anteriores

Interfaces funcionais: Function

```
1 package java.util.function;
2
3 public interface Function<T,R> {
4     public R apply(T t);
5 }
```

```
15      List<SalesTxn> tList = SalesTxn.createTxnList();
16      SalesTxn first = tList.get(0);
17      Function<SalesTxn, String> buyerFunction =
18          t -> t.getBuyer().getName();
19
20      System.out.println("\n== First Buyer");
21      System.out.println(buyerFunction.apply(first));
22  }
```

Interfaces funcionais: Binary types

```
1 package java.util.function;
2
3 public interface BiPredicate<T, U> {
4     public boolean test(T t, U u);
5 }
```

```
14     List<SalesTxn> tList = SalesTxn.createTxnList();
15     SalesTxn first = tList.get(0);
16     String testState = "CA";
17
18     BiPredicate<SalesTxn,String> stateBiPred =
19       (t, s) -> t.getState().getStr().equals(s);
20
21     System.out.println("\n== First is CA?");
22     System.out.println(stateBiPred.test(first, testState));
```

Streams

- Streams:
  - java.util.stream
  - Sequência de elementos na qual vários métodos podem ser encadeados
  - Transformam as collections em pipelines de execução

- Características:
  - Imutáveis
  - Após os elementos terem sido consumidos, já não “existem” na stream
  - Uma cadeia de operações pode ser executada uma única vez em cada stream
  - Podem ser executadas em série ou em paralelo
  - São lazy, ou seja, executam apenas as operações estritamente necessárias

- Uma stream (pipeline) consiste em:
  - Uma fonte de objetos (uma collection)
  - Zero ou mais operações intermédias (filter, map, peek)
  - Uma operação terminal (forEach, sum, count, average, min, max)
  - Ou uma operação terminal short-circuit (findFirst, findAny, anyMatch)

Streams

- As operações intermédias (filtros, por exemplo) usam expressões lambda para escolher os items:

```
tList.stream()
    .filter(t -> t.getState().equals("CA"))
    .forEach(SalesTxn::printSummary);
```

- Referências a métodos:
  - Por vezes a expressão lambda apenas tem uma linha/chamada a um método:
    .forEach(t -> t.printSummary())
  - Alternativamente podemos usar referências para métodos:
    .forEach(SalesTxn::printSummary));
  - Podemos usar referências para métodos nas seguintes situações:
    - Método estático
      .forEach(SalesTxn::printSummary));
    - Referência a um método da instância
    - Referência a um método da instância dum objeto arbitrário dum tipo específico
      String::compareToIgnoreCase
    - Construtor
      ClassName::new

Streams

- Os pipelines permitem encadeamento de métodos
- Os métodos podem incluir filtros e outros

```
tList.stream()
    .filter(t -> t.getState().equals("CA"))
    .filter(t -> t.getBuyer().getName().equals("AcmeElectronics"))
    .forEach(SalesTxn::printSummary);
```

- Podem encadear-se expressões lógicas:

```
tList.stream()
    .filter(t -> t.getState().equals("CA") &&
        t.getBuyer().getName().equals("Acme Electronics"))
    .forEach(SalesTxn::printSummary);

tList.stream()
    .filter(t -> t.getState().equals("CA"))
    .filter(t -> t.getBuyer().getName().equals("Acme Electronics"))
    .forEach(SalesTxn::printSummary);
```

Filtrar dados

`filter(Predicate<? super T> predicate)`

- Filtra os dados de uma stream mantendo apenas os elementos cuja execução do predicate devolva true

```
List<String> stringList = new ArrayList<>();

stringList.add("43");
stringList.add("11");
stringList.add("8");
stringList.add("743");

stringList.stream()
    .filter(s -> s.length() == 2)
    .forEach(System.out::println);
```

```
43
11
```

Creating Constructors

`map(Function<? super T,? extends R> mapper)`

- Um mapa tranforma os elementos de uma Stream naquilo que nós quisermos (daí receber uma Function como parâmetro)
- Um map recebe, como argumento, uma `Function`
  - Uma `Function` recebe um genérico como argumento e devolve um outro
  - Versões primitivas do map
    - `mapToInt()` `mapToLong()` `mapToDouble()`

```
List<String> stringList = new ArrayList<>();

stringList.add("43");
stringList.add("11");
stringList.add("6");
stringList.add("743");
stringList.add("23");

stringList.stream()
    .map(t -> {
        Integer i = Integer.valueOf(t);
        return i + 5;
    })
    .forEach(System.out::println);
```

```
48
16
11
748
28
```

Peek

`peek(Consumer<? super T> action)`

- O método peek executa a operação indicada pela expressão lâmbda e devolve o elemento à stream
- Apropriado para, por exemplo, apresentar resultados intermédios

```
List<String> stringList = new ArrayList<>();

stringList.add("43");
stringList.add("11");
stringList.add("6");
stringList.add("743");
stringList.add("23");

stringList.stream()
    .peek(t -> {
        int i = Integer.valueOf(t).intValue();
        if(i > 100) {
            i += 5;
            System.out.println("Dentro do peek: " + i);
        }
    })
    .forEach(System.out::println);
```

```
43
11
6
Dentro do peek: 748
743
23
```

Operações terminais Short-circuit

`findFirst()`

- Devolve o primeiro elemento da stream

```
List<String> stringList = new ArrayList<>();

stringList.add("43");                                43
stringList.add("11");

stringList.stream().findFirst().ifPresent(System.out::println);
```

`allMatch(Predicate<? super T> predicate)`

- Devolve true se todos os elementos da stream fizerem match com a condição apresentada

```
List<String> stringList = new ArrayList<>();

stringList.add("43");
stringList.add("11");                    true

boolean under50 =
    stringList.stream()
        .allMatch(t -> Integer.valueOf(t) < 50);
System.out.println(under50);
```

`noneMatch(Predicate<? super T> predicate)`

- Devolve true se nenhum dos elementos da stream fizer match com a condição apresentada

```
List<String> stringList = new ArrayList<>();

stringList.add("43");
stringList.add("11");                false

boolean noneOver40 =
    stringList.stream()
        .noneMatch(t -> Integer.valueOf(t) > 40);
System.out.println(noneOver40);
```

Operações terminais Short-circuit

`findFirst()`

- Devolve o primeiro elemento da stream

```
List<String> stringList = new ArrayList<>();

stringList.add("43");                                    43
stringList.add("11");

stringList.stream().findFirst().ifPresent(System.out::println);
```

`allMatch(Predicate<? super T> predicate)`

- Devolve true se todos os elementos da stream fizerem match com a condição apresentada

```
List<String> stringList = new ArrayList<>();

stringList.add("43");
stringList.add("11");                    true

boolean under50 =
    stringList.stream()
        .allMatch(t -> Integer.valueOf(t) < 50);
System.out.println(under50);
```

`noneMatch(Predicate<? super T> predicate)`

- Devolve true se nenhum dos elementos da stream fizer match com a condição apresentada

```
List<String> stringList = new ArrayList<>();

stringList.add("43");
stringList.add("11");                false

boolean noneOver40 =
    stringList.stream()
        .noneMatch(t -> Integer.valueOf(t) > 40);
System.out.println(noneOver40);
```

Operações terminais Short-circuit

anyMatch(Predicate<? super P_OUT> predicate)

- Devolve true se pelo menos um dos elementos da stream fizer match com a condição apresentada

```
List<String> stringList = new ArrayList<>();

stringList.add("43");
stringList.add("11");
stringList.add("6");
stringList.add("743");

System.out.println(stringList.stream()
        .anyMatch(t -> {
            int i = Integer.valueOf(t).intValue();
            return i > 40;
        }));
```

```
true
```

Operações terminais

`count()`

- Devolve o número de elementos da stream

```
List<String> stringList = new ArrayList<>();

stringList.add("43");
stringList.add("11");
stringList.add("6");
stringList.add("743");

System.out.println(stringList.stream()
        .filter(t -> t.contains("4"))
        .count());
```

2

`max(Comparator<? super T> comparator)`

- Devolve o elemento máximo

```
List<String> stringList = new ArrayList<>();

stringList.add("43");
stringList.add("11");
stringList.add("6");
stringList.add("743");

System.out.println(stringList.stream()
        .filter(t -> t.contains("4"))
        .max(Comparator.naturalOrder()).get());
```

743

`min(Comparator<? super T> comparator)`

- Devolve o elemento mínimo

Operações terminais

`average()`

- Só se aplica a streams numéricas (`DoubleStream`, `IntStream` ou `LongStream`)
- Devolve um Optional com a média aritmética dos valores desta stream

```
List<String> stringList = new ArrayList<>();

stringList.add("43");
stringList.add("11");
stringList.add("6");
stringList.add("743");

System.out.println(stringList.stream()
        .mapToInt(Integer::valueOf)
        .average().getAsDouble());
```

```
200.75
```

`sum()`

- Só se aplica a streams numéricas (`DoubleStream`, `IntStream` ou `LongStream`)
- Devolve um valor do tipo da stream com a soma dos valores desta stream

```
List<String> stringList = new ArrayList<>();

stringList.add("43");
stringList.add("11");
stringList.add("6");
stringList.add("743");

System.out.println(stringList.stream()
        .mapToLong(Long::valueOf)
        .sum());
```

```
803
```

Collections

- *Generics*
- *Data Structures*

Generics

- Generics são uma funcionalidade do Java que foi introduzida na versão 5
- Permitem tornar o código mais **genérico**, por um lado, e fazer um número maior de **validações de tipos em compile time**, por outro.
- Os Generics permitem que o programador escreva código para **um tipo que consiste numa variável**. Assim sendo, o mesmo código, mediante parametrização, **pode aplicar-se a qualquer tipo** da mesma forma
- Esta parametrização permite fazer a **validação de tipos em compile time** sem necessitar que sejam escritas classes específicas para um determinado tipo
- **Antes dos generics**, era usada a classe **Object** como alternativa. No entanto, esta alternativa tem **problemas de type safety**

Object sem type safety

```
class BoxPrinter {
        private Object val;
        public BoxPrinter(Object arg) {
                val = arg;
        }
        public String toString() {
                return "[" + val + "]";
        }
        public Object getValue() {
                return val;
        }
}

class BoxPrinterTest2 {
        public static void main(String []args) {
                BoxPrinter value1 = new BoxPrinter(new Integer(10));
                System.out.println(value1);
                Integer intValue1 = (Integer) value1.getValue();

                BoxPrinter value2 = new BoxPrinter("Hello world");
                System.out.println(value2);
                        // OOPs! by mistake, we did (Integer) cast instead of (String)
                        Integer intValue2 = (Integer) value2.getValue();
        }
}
```

Como é usado o tipo Object, não há validação de tipos. Assim sendo, o erro que existe/ocorre neste código só é detetavel em run time, o que os torna muito mais complicados de corrigir

Here is the output:

```
[10]
[Hello world]
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to
java.lang.Integer at BoxPrinterTest2.main(Main.java:22)
```

In the line

```
Integer intValue2 = (Integer) value2.getValue();
```

Generics com type safety

```
// This program demonstrates the type-safety feature of generics
class BoxPrinter<T> {
        private T val;
        public BoxPrinter(T arg) {
                val = arg;
        }
        public String toString() {
                return "[" + val + "]";
        }
        public T getValue() {
                return val;
        }
}

class BoxPrinterTest4 {
        public static void main(String []args) {
                BoxPrinter<Integer> value1 = new BoxPrinter<Integer>(new Integer(10));
                System.out.println(value1);
                Integer intValue1 = value1.getValue();

                BoxPrinter<String> value2 = new BoxPrinter<String>("Hello world");
                System.out.println(value2);
                // OOPs! by mistake, we did put String in an Integer
                Integer intValue2 = value2.getValue();
        }
}
```

This is the line where you made a mistake and tried to put a String in an Integer:

```
Integer intValue2 = value2.getValue();
```

And you get the following compiler error:

```
BoxPrinterTest.java:23: incompatible types
found   : java.lang.String
required: java.lang.Integer
                Integer intValue2 = value2.getValue();
```

A utilização de generics permite que os tipos sejam parametrizados dando, assim, possibilidade ao compilador de validar os tipos de dados em compile time (o que facilita a deteção e correção de erros)

Como criar e usar generics

```
// Esta classe mostra como usar Generics
public class Pair<T1, T2> {
    private T1 object1;
    private T2 object2;

    public Pair(T1 object1, T2 object2) {
        this.object1 = object1;
        this.object2 = object2;
    }

    public T1 getFirst() {
        return object1;
    }

    public T2 getSecond() {
        return object2;
    }
}
```

A partir do Java 7 pode usar-se a diamond notation, onde o construtor assume implicitamente os generics da variável declarada:

```
Pair<String, Integer> pair =
        new Pair<>("Hello world", Integer.valueOf(5));
```

```
Pair<String, Integer> pair = new Pair<String, Integer>("Hello world", Integer.valueOf(5));

pair.get
```

```
getClass() : Class<?> - Object
getFirst() : String - Pair
getSecond() : Integer - Pair
```

Métodos genéricos

- Tal como com classe genéricas, podemos também criar métodos genéricos, ou seja, métodos que podem receber parâmetros de tipos genéricos (a classe java.util.Collections, por exemplo, tem numerosos métodos genéricos)

```
public static <T> boolean addElementToList(List<T> list, T item) {
    return list.add(item);
}
```

- Ao invocar o método, na maior parte dos casos, o compilador pode inferir os tipos corretos para atribuir aos generics:

```
List<String> list = new ArrayList<>();
String ola = "Hello world!";

addElementToList(list, ola);
```

- Quando não se conseguem inferir os tipos, como no seguinte exemplo (mesmo aqui o compilador consegue inferir o tipo se o resultado da chamada deste método for atribuído explicitamente a uma variável com um tipo correto):

```
public <T extends Animal> T callFriend(String name) {
```

- Pode passar-se explicitamente o parâmetro do tipo da seguinte forma:

```
jerry.<Dog>callFriend("spike").bark();
```

Genéricos, os tipos e o type erasure

- Ao contrário dos restantes elementos da linguagem Java, os generics não permitem herança nem conversão de tipos:

```
List<Integer> integerList = new ArrayList<>();

List<Number> numberList = integerList;
```

- Uma razão para esse comportamento é a seguinte (caso a 1ª linha fosse permitida:

```
List<Number> numberList = new ArrayList<Integer>();
numberList.add(Integer.valueOf(10)); // OK
numberList.add(Float.valueOf(10.0f)); // ERROR!!!
```

- A outra razão é o type erasure (o que não permite que haja tipos dinâmicos associados aos generics):
  - Durante a compilação os tipos são validados e verificados
  - Após estas verificações, ocorre o type erasure que remove toda a informação sobre os generics das classes
  - Após esta ação, as classes são deixadas tal como eram antes dos generics

```
public class Pair<T1, T2> {
    private T1 object1;
    private T2 object2;

    public Pair(T1 object1, T2 object2) {
        this.object1 = object1;
        this.object2 = object2;
    }

    public T1 getFirst() {
```

```
public class Pair {
    private Object object1;
    private Object object2;

    public Pair(Object object1, Object object2) {
        this.object1 = object1;
        this.object2 = object2;
    }

    public Object getFirst() {
```

Generics e wildcards

- Para resolver o problema de inexistência de herança e conversão de tipos nos generics e para permitir fazer código mais genérico, foi adicionada a possibilidade de existirem wildcards em generics

```
List<?> listWildcard =
        new ArrayList<Integer>();
```

```
List<Object> listObject =
        new ArrayList<Integer>();
```

- No entanto existem alguns problemas com esta solução.

```
List<?> listWildcard =
        new ArrayList<Integer>();

listWildcard.add(Integer.valueOf(45));
System.out.println(listWildcard);
```

The method add(capture#1-of ?) in the type List<capture#1-of ?> is not applicable for the arguments (Integer)

- O wildcard (?) significa “tipo desconhecido”. Não é permitido fazer uma atribuição de uma variável de um qualquer tipo a uma variável de “tipo desconhecido” (objetivo primordial dos generics)
- É, no entanto, possível atribuir uma variável de “tipo desconhecido” a uma variável do tipo Object.

```
List<?> listWildcard =
        new ArrayList<Integer>();

// codigo...

Object o = listWildcard.get(4);
System.out.println("Valor: " + o);
```

Bounded Wildcards

- Agora que temos a liberdade de usar qualquer tipo num determinado generic, seria interessante termos alguma forma de limitar esta liberdade, por exemplo através da hierarquia de classes
- Surgem assim os bounded wildcards:

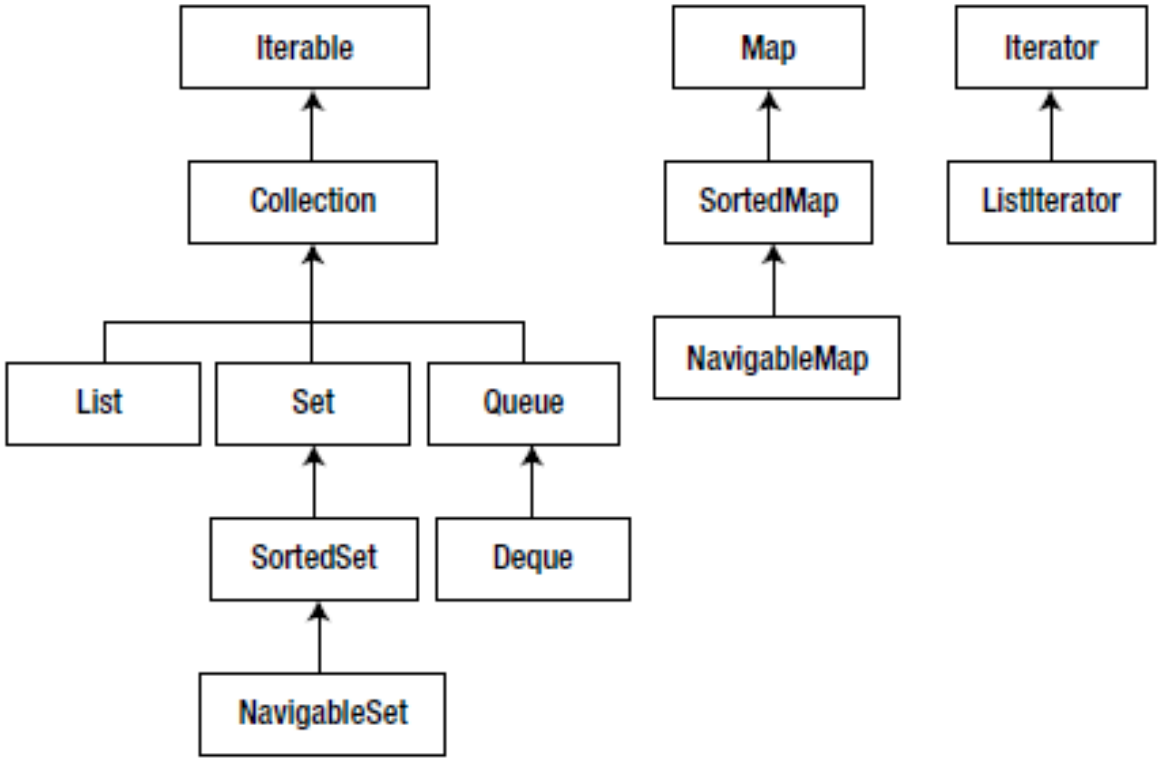
```
List<? super Number> superNumberList        = new ArrayList<Number>();
List<? extends Number> extendsNumberList    = new ArrayList<Number>();
// List<? super Number> integerSuperList    = new ArrayList<Integer>(); // Integer não é superclasse de Number
List<? extends Number> integerExtendsList   = new ArrayList<Integer>();
List<? super Number> objectSuperList        = new ArrayList<Object>();
// List<? extends Number> objectExtendsList = new ArrayList<Object>(); // Object não é subclasse de Number

superNumberList.add(Integer.valueOf(3));
// Number num = superNumberList.get(1); // Só pode devolver Object
Object obj = superNumberList.get(1);
// extendsNumberList.add(Integer.valueOf(3)); // Não é possivel garantir que um Integer será suportado
Number num = extendsNumberList.get(1);
```

Java Collections Framework

- A Java collections framework fornece um conjunto amplo de estruturas de dados e algoritmos já prontos a usar
- Estas funcionalidades podem ser usadas com qualquer tipo de dados de uma forma “type-safe” devido ao uso extensivo de generics
- Esta framework tem três tipos de componentes principais:
  - **Classes abstratas e interfaces**: Fornecem funcionalidade geral. Compreendendo-as entendemos as funcionalidades do contrato
  - **Classes concretas**: São as reais implementações dos containers. Implementam as funcionalidades específicas, mas, normalmente, não acrescentam conceitos novos às classes abstratas/interfaces correspondentes
  - **Algoritmos**: A classes java.util.Collections implementa funcionalidade comum, como por exemplo ordenação, pesquisa, etc.

java.util interfaces



java.util interfaces

- **Iterable**: Interface introduzida no java 5 que é a base de todos os objetos que podem ser usados num foreach
- **Collection**: Interface mais genérica de todos os containers que guardam elementos únicos (em oposição aos mapas que guardam elementos binários, ou seja, pares chave-valor). Estes containers permitem adicionar e remover elementos, contar, transformar em array, limpar, verificar se os elementos existem no container, etc.
- **List**: Containers que estendem a Collection e cujos elementos:
  - Têm uma ordem (ou de introdução ou uma ordem que o programador definir)
  - Podem conter elementos repetidos

java.util interfaces

- **Set**: Containers que estendem a Collection e cujos elementos:
  - Não têm uma ordem definida
  - Não podem conter elementos repetidos
- **Queue**: Containers que estendem a Collection, representam uma fila de espera e cujos elementos:
  - São ordenados pela ordem de entrada
  - Podem conter elementos repetidos
- **Map**: Mapas são containers associativos (portanto não estendem a collection), onde não é permitido existirem elementos com a mesma chave
- **Iterator**: Objetos que percorrem sequencialmente os itens das collections

Listas

- Vemos ver aqui duas implementações de listas disponibilizadas pela Java Collections Framework:
  - **ArrayList**: Mistura os benefícios de um array e de uma lista.
    - A implementação baseia-se num **array**
    - O tempo de **acesso** aos elementos é **constante**
    - O tempo de **adicionar e remover** elementos pode ser **grande** (linear)
    - **Tunning**: dimensão inicial do array (no construtor)
  - **LinkedList**: Implementação tradicional de lista
    - Conjunto de nós com o elemento e mais a **referência para o próximo nó**
    - Tempo de **pesquisa** dos elementos pode ser **grande** (linear)
    - Tempo de **adicionar ou remover** elementos é **constante**
    - Implementa a **interface Deque** (double ended queue) fornecendo assim funcionalidades de filas **FIFO e LIFO**
    - Não tem parâmetros de tunning

Listas

```java
List<String> list1 = new ArrayList<>();

list1.add("Ola"); // true
list1.add("Ole"); // true
String addedText = list1.get(0); // "Ola"
String addedText2 = list1.remove(0); // "Ola"
addedText.equals(addedText2); // true
list1.size(); // 1
list1.contains(addedText); // false

for (String item : list1) {
    System.out.println(item);
}

Iterator<String> iterator = list1.iterator();
while(iterator.hasNext()) {
    String item = iterator.next();
    System.out.println(item);
    iterator.remove();
}

list1.clear(); // limpa a lista, embora já esteja limpa
list1.isEmpty(); // true
```

Maps

- A Java Collections Framework disponibiliza três implementações de maps:
  - **HashMap:**
    - A implementação baseia-se num array de Entries (chave-valor-próximo)
    - É a mais rápida das três implementações
    - Não oferece ordenação
    - **Tunning:** dimensão inicial e load factor
  - **TreeMap:**
    - A implementação baseia-se em Entries que referenciam as outras Entries num esquema em árvore
    - É a mais lenta das três implementações
    - Oferece ordenação
    - Não tem parâmetros de tunning
  - **LinkedHashMap:**
    - Baseia-se na implementação de HashMap
    - É ligeiramente mais lenta que o HashMap por causa ligação aos outros items
    - Oferece a ordenação de introdução inicial ou por acesso
    - Permite remover os elementos acedidos há mais tempo
    - **Tunning**: dimensão inicial e load factor

Maps

```
Map<Integer, String> map1 = new HashMap<>();

map1.put(Integer.valueOf(1), "Ola"); // null
map1.put(Integer.valueOf(2), "Ole"); // null
map1.put(Integer.valueOf(3), "Ola"); // null
map1.size(); // 3
map1.remove(Integer.valueOf(1)); // "Ola"
map1.containsKey(Integer.valueOf(2)); // true
map1.containsValue("Ole"); // true

Set<Entry<Integer, String>> entrySet = map1.entrySet();
for (Entry<Integer, String> entry : entrySet) {
    System.out.println(entry.getKey() + ":" + entry.getValue());
} // "2:Ole" / "3:Ola"

map1.isEmpty(); // false
```

Sets

- A Java Collections Framework disponibiliza três implementações de sets:
  - **HashSet**:
    - A implementação baseia-se num HashMap
    - É a mais rápida das três implementações
    - Não oferece ordenação
    - **Tunning**: dimensão inicial e load factor
  - **TreeSet**:
    - A implementação baseia-se num TreeMap
    - É a mais lenta das três implementações
    - Oferece ordenação
    - Não tem parâmetros de tunning
  - **LinkedHashSet**:
    - A implementação baseia-se num LinkedHashMap
    - É ligeiramente mais lenta que o HashSet por causa ligação aos outros items
    - Apenas oferece a ordenação de introdução inicial
    - **Tunning**: dimensão inicial e load factor

Sets

```
Set<String> set1 = new HashSet<>();

set1.add("Ola"); // true
set1.add("Ole"); // true
set1.add("Ola"); // false
set1.size(); // 2
set1.remove("Ole"); // true
set1.contains("Ola"); // true

Iterator<String> iterator = set1.iterator();
while(iterator.hasNext()) {
    String item = iterator.next();
    System.out.println(item);
    iterator.remove();
}

set1.isEmpty(); // true
```

```
NavigableSet<String> set1 = new TreeSet<>();

set1.add("Ola"); // true
set1.add("Ole"); // true
set1.add("Oli"); // true

set1.first(); // "Ola"
set1.last(); // "Oli"
set1.ceiling("Olb"); // "Ole"
set1.floor("Olb"); // "Ola"

Iterator<String> iterator = set1.descendingIterator();
while(iterator.hasNext()) {
    String item = iterator.next();
    System.out.println(item);
}
// "Oli" / "Ole" / "Ola"

set1.isEmpty(); // false
```

Luís Ribeiro

I'm a software engineer with more than 14 years of experience in software development in Java and other technologies, software architecture, team management and project management.

My mission is to transform people's ideas into fully functional, production ready and user friendly software applications that can change the world!

luismmribeiro@gmail.com

Thank you