Apontamentos de Estruturas de Dados

Plataforma de Coleções do JAVA e Genéricos

Ricardo Santos| rjs@estg.ipp.pt

Escola Superior de Tecnologia e Gestão
Instituto Politécnico do Porto

Última versão: Outubro de 2022

Índice

1. Plataforma de Coleções do JAVA

Qualquer grupo de objetos individuais representados como uma única entidade é um conceito conhecido como coleção de objetos. Em Java, existe desde a versão *Java Development Kit* (JDK) 1.2, uma plataforma denominada de Plataforma de Coleções que contém todas as classes da coleção assim como as interfaces a serem implementadas.

A interface `Collection` (`java.util.Collection`) e a interface `Map` (`java.util.Map`) são as duas principais interfaces e a "raiz" das classes das várias coleções. Claro que para se compreender melhor o que é a Plataforma de Coleções temos primeiro de compreender o que é neste contexto uma plataforma. Uma plataforma (ou *framework*) é um conjunto de classes e interfaces que fornecem uma arquitetura pré-definida, neste caso em específico, para gerir coleções. As coleções por sua vez, são formas de organizar grupos de "coisas" e manipular esses grupos como entidades únicas.

Antes da *Collection Framework* (ou antes do JDK 1.2) ser introduzida, os métodos padrão para agrupar objetos Java (ou coleções) eram realizados através de `Arrays` (ou `Vectors`) ou `Hashtables`, no entanto essas coleções não tinham nenhuma interface comum. Embora o objetivo principal de todas essas coleções fosse o mesmo, a implementação de todas essas coleções foi definida de forma independente e não havia correlação entre elas. Esta abordagem tornou o cenário da utilização destas coleções por parte dos clientes bastante mais complexo já que teriam de se lembrar de todos os diferentes métodos e construtores presentes em cada classe.

Para reduzir os problemas mencionados foi decidido criar uma interface que foi introduzida na *Collection Framework* da versão 1.2 do JDK. A introdução desta plataforma e a sua utilização trouxe as seguintes vantagens para toda a comunidade:

1. **API consistente**: A API tem um conjunto básico de interfaces (`Collection`, `Set`, `List` ou `Map`, etc) e todas as classes (`ArrayList`, `LinkedList`, etc) que implementam essas interfaces têm um conjunto comum de métodos.

2. **Reduz o esforço de programação**: um programador não precisa preocupar-se com a definição da coleção, deve concentrar-se especificamente na sua escolha para uma melhor utilização no contexto de um programa. Portanto, o conceito básico de programação orientada a objetos (ou seja, abstração) foi implementado com sucesso.

3. **Aumenta a velocidade e a qualidade geral do programa**: Aumenta o desempenho através das implementações de alto desempenho de estruturas de dados e algoritmos úteis porque os programadores não precisam pensar na melhor implementação de uma determinada estrutura de dados. Deve simplesmente usar a melhor implementação para aumentar drasticamente o desempenho do seu algoritmo/programa.

O pacote utilitário `java.util` contém todas as classes e interfaces exigidas pela plataforma de coleções. Todas essas interfaces e classes fornecem métodos de inserção, remoção, pesquisa, ordenação e manipulação de elementos. É de notar também que algumas coleções permitem apenas objetos únicos, por exemplo `Sets`, enquanto outras estruturas permitem duplicações como `ArrayLists`. Cada coleção pode trazer um ganho de produtividade se utilizada corretamente. É importante salientar que a interface `Map` também está presente na plataforma de coleções apesar de não estar na hierarquia de `Collection`. É portanto uma interface base fora da hierarquia da `Collection`. A figura 1 ilustra a hierarquia da plataforma de coleções.

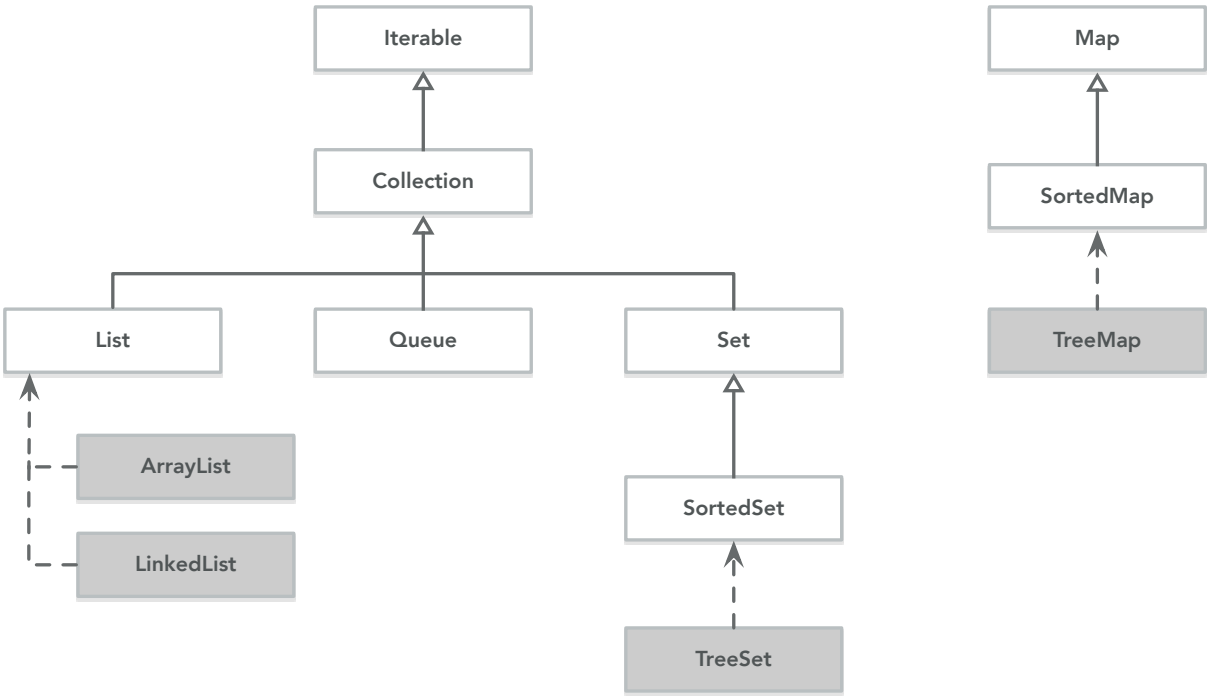

Figura 1 – Hierarquia da Plataforma de Coleções Java

Antes de entender os diferentes componentes da plataforma de coleções é necessário relembrar a seguinte terminologia:

- Interface - As interfaces são declaradas com recurso à palavra reservada `interface` e podem conter apenas assinaturas de métodos e declarações de constantes;
- Classe abstrata - É uma classe que possui pelo menos um método abstrato, isto é, não pode ser instanciada;
- Classe concreta - É uma classe que pode ter instâncias.

2. Genéricos

Em qualquer projeto de software não trivial, os *bugs* são simplesmente uma realidade da qual não conseguimos escapar. Planeamento, programação e testes cuidadosos podem ajudar a reduzir a sua existência, mas de alguma forma irão sempre aparecer no nosso *software*. À medida que o nosso código cresce tanto em tamanho como em complexidade, fica cada vez mais evidente a necessidade de adotarmos melhores práticas e métodos para que o nosso trabalho seja mais eficiente e apresente melhor qualidade. Existem vários tipos de *bugs*, uns mais fáceis que outros de detetar:

- *Bugs* em tempo de compilação - por exemplo informam o programador imediatamente que algo está errado; podemos simplesmente usar as mensagens de erro do compilador para descobrir qual é o problema e corrigi-lo;
- *Bugs* em tempo de execução - podem ser muito mais problemáticos; nem sempre aparecem imediatamente e, quando o fazem, pode ser num local muito distante do local exato do problema.

Os genéricos adicionam estabilidade ao código permitindo que grande parte dos *bugs* sejam detetados em tempo de compilação evitando problemas maiores no futuro. Os genéricos são um tópico muito importante na programação orientada a objetos e fundamentais para a aprendizagem de estruturas dados com a plataforma de coleções já que a maior parte das classes faz uso de genéricos.

2.1. Exemplo de uma Simples Caixa

Vamos começar por implementar uma classe `Box` não genérica que usa objetos de qualquer tipo. A classe possui apenas duas operações: `add`, que adiciona um objeto à caixa e `get` que obtém um objeto da caixa.

```
1. public class Box {
2.     private Object object;
3.     public void add(Object object) {
```

```
4.        this.object = object;
5.    }
6.    public Object get() {
7.        return object;
8.    }
9. }
```

Figura 2 – Classe Box

Como os métodos aceitam ou retornam `Object` podemos passar o que quisermos, desde que não seja um dos tipos primitivos. No entanto, se precisarmos restringir o tipo contido a algo específico (como `Integer`), a única opção seria especificar o requisito na documentação (ou neste caso, um comentário), sobre o qual o compilador obviamente não sabe nada:

```
1. public class BoxDemo1 {
2. public static void main(String[] args) {
3.        // ONLY place Integer objects into this box!
4.        Box integerBox = new Box();
5.        integerBox.add(new Integer(10));
6.        Integer someInteger = (Integer)integerBox.get();
7.        System.out.println(someInteger);
8.   }
```

Figura 3 - Classe BoxDemo1

O programa `BoxDemo1` (figura 3) cria um objeto `Integer`, passa-o para o `add` (linha 5) e atribui esse mesmo objeto a `someInteger` pelo valor de retorno de `get` (linha 6). Em seguida, imprime o valor do objeto (`10`) para o standard output (linha 7). Sabemos que a conversão de `Object` para `Integer` está correta porque honramos o "contrato" especificado no comentário. Mas lembre-se, o compilador não sabe nada acerca disso – o compilador confia que a nossa conversão está correta. Além disso, não fará nada para evitar que um programador descuidado passe um objeto do tipo errado, como por exemplo uma `String`:

```
1. public class BoxDemo2 {
2.     public static void main(String[] args) {
3.         // ONLY place Integer objects into this box!
4.         Box integerBox = new Box();
5.         // Imagine this is one part of a large application
6.         // modified by one programmer.
7.         integerBox.add("10"); // Note how the type is now String
8.         // ... and this is another, perhaps written
9.         // by a different programmer
10.        Integer someInteger = (Integer)integerBox.get();
11.        System.out.println(someInteger);
12.    }
13.}
```

Figura 4 - Classe BoxDemo2

No exemplo `BoxDemo2` (figura 4), armazenamos o número `10` como uma `String` (linha 10), o que pode ser o caso quando, digamos, uma interface gráfica (GUI) armazena o valor de entrada de um determinado utilizador. No entanto, a conversão existente de `Object` para `Integer` irá falhar. Este é um problema muito comum que passa facilmente despercebido já que o código compila e é impossível sabermos qual o problema até chegarmos ao tempo de execução. Este problema irá resultar numa exceção do tipo `ClassCastException`:

```
1. Exception in thread "main" java.lang.ClassCastException:
   java.lang.String cannot be cast to java.lang.Integer at
   BoxDemo2.main(BoxDemo2.java:7)
```

Figura 5 - Exceção na classe BoxDemo2

Se a classe `Box` tivesse sido pensada desde o início com genéricos, este erro teria sido detectado pelo compilador em vez de "crashar" o programa em tempo de execução.

2.2. Tipos Genéricos

Vamos atualizar nossa classe `Box` para usar genéricos. Primeiro, criamos uma declaração de tipo genérico alterando o código "`public class Box`" para "`public class Box<T>`"; esta definição introduz uma variável de tipo, denominada `T`, que pode ser usada em qualquer lugar dentro da classe. Essa mesma técnica também pode ser aplicada a interfaces. Não há nada particularmente complexo sobre este conceito. Na verdade, é bem parecido com o que já sabemos acerca de variáveis de uma forma geral. Devemos apenas pensar em T como um tipo especial de variável cujo "valor" será qualquer tipo que pretendermos passar; pode ser qualquer tipo de classe, qualquer tipo de interface ou até mesmo outra variável de tipo. Só não podemos passar tipos de dados primitivos para `T`, essa é a única limitação. Neste contexto, também dizemos que `T` é um parâmetro de tipo formal da classe `Box`.

```
1. /**
2. * Generic version of the Box class.
3. */
4. public class Box<T> {
5.    private T t; // T stands for "Type"
6.    public void add(T t) {
7.       this.t = t;
8.    }
9.    public T get() {
10.      return t;
11.   }
12.}
```

Figura 6 - Classe `Box` genérica

Repare que substituímos todas as ocorrências de `Object` por `T`. Para fazer referência a esta classe genérica devemos executar uma invocação de tipo genérico que substitui `T` por algum valor concreto como por exemplo `Integer`:

```
1. Box<Integer> integerBox;
```

Figura 7 - Declaração de `integerBox` através de um tipo genérico

Podemos pensar numa invocação de tipo genérico como sendo semelhante a uma simples invocação de um método, mas em vez de passarmos um argumento passamos um tipo — `Integer` neste caso — para a própria classe `Box`. Como qualquer outra declaração de variável este código não cria um novo objeto `Box`. Declara simplesmente que `integerBox` irá manter uma referência para uma "`Box` de `Integer`", que é como `Box<Integer>` é lido.

A invocação de um tipo genérico é geralmente referida como um tipo parametrizado. Para instanciar esta classe, use a palavra-chave `new` como de costume, mas coloque `<Integer>` entre o nome da classe e os parênteses:

```
1. integerBox = new Box<Integer>();
```

Figura 8 - Instanciação de `integerBox` através de um tipo genérico

Ou podemos colocar simplesmente a instrução inteira numa linha como:

```
1. Box<Integer> integerBox = new Box<Integer>();
```

Figura 9 - Declaração e Instanciação de `integerBox` através de um tipo genérico

Após `integerBox` ser inicializado podemos simplesmente invocar o método `get` sem sermos obrigados a fazer uma conversão como no exemplo da figura 4 (`BoxDemo2`), como em `BoxDemo3` (figura 10):

```
1. public class BoxDemo3 {
2.     public static void main(String[] args) {
3.         Box<Integer> integerBox = new Box<Integer>();
4.         integerBox.add(new Integer(10));
5.         Integer someInteger = integerBox.get(); // No cast!
6.         System.out.println(someInteger);
7.     }
8. }
```

Figura 10 - Classe `BoxDemo3`

Além disso, podemos tentar adicionar um tipo incompatível à `Box` como por exemplo `String`, nesse caso a compilação irá falhar alertando para o que anteriormente seria um *bug* em tempo de execução:

```
1. BoxDemo3.java:5: add(java.lang.Integer) in Box<java.lang.Integer>
   cannot be applied to (java.lang.String) integerBox.add("10");
2.                                                     ^
3. 1 error
```

Figura 11 - Exceção ao adicionar tipo incompatível à `Box`

É importante entender que as variáveis de tipo não são realmente tipos. Nos exemplos anteriores não iremos encontrar `T.java` ou `T.class` em qualquer lugar do nosso sistema de ficheiros. Além disso, `T` não faz parte do nome da classe `Box`. Na verdade, durante a compilação todas as informações genéricas serão removidas por completo deixando apenas `Box.class` no sistema de ficheiros. Observe também que um tipo genérico pode ter vários parâmetros de tipo mas cada parâmetro deve ser exclusivo dentro da classe ou interface em que é declarado. Uma declaração de `Box<T,T>`, por exemplo, irá gerar um erro na segunda ocorrência de `T`, mas `Box<T,U>`, no entanto, é permitido.

2.3. Convenções de Nomenclatura para o Tipo Parametrizado

Por convenção, os nomes dos tipos parametrizados são letras maiúsculas simples. Isso contrasta fortemente com as convenções de nomenclatura de variáveis que já conhecemos, e por um bom motivo: sem essa convenção seria difícil dizer a diferença entre uma variável de tipo e uma classe comum ou nome de interface.

Os nomes de tipos parametrizados mais usados são:

- E – Elemento (usado extensivamente pela *Java Collections Framework*)
- K – Chave (key)
- N – Número
- T – Tipo
- V – Valor

- S,U,V etc. – 2º, 3º, 4º tipos.

2.4. Métodos e Construtores Genéricos

Os tipos parametrizados também podem ser declarados dentro de assinaturas de método ou construtor para criar métodos genéricos e construtores genéricos. Esta possibilidade é semelhante a declarar um tipo genérico, mas o escopo do tipo parametrizado é limitado ao método ou construtor no qual ele é declarado.

```
1. /**
2. * This version introduces a generic method.
3. */
4. public class Box<T> {
5.
6.    private T t;
7.
8.    public void add(T t) {
9.       this.t = t;
10.   }
11.
12.   public T get() {
13.      return t;
14.   }
15.
16.   public <U> void inspect(U u){
17.      System.out.println("T: " +t.getClass().getName());
18.      System.out.println("U: " +u.getClass().getName());
19.   }
20.
21.   public static void main(String[] args) {
22.      Box<Integer> integerBox = new Box<Integer>();
23.      integerBox.add(new Integer(10));
24.      integerBox.inspect("some text");
25.   }
26.}
```

Figura 12 - Classe Box com método genérico

Aqui, adicionamos um método genérico denominado de `inspect`, que define um tipo parametrizado `U`. Este método aceita um objeto e imprime o seu tipo no *standard output*. Para comparação também imprime o tipo de `T`. Por conveniência a classe também possui um método `main` para que possa ser executada.

O *output* deste programa é:

```
1. T: java.lang.Integer
2. U: java.lang.String
```

Figura 13 - *Output* da figura 12

Ao passar diferentes tipos o *output* será alterado de acordo.

Um uso mais realista de métodos genéricos pode ser algo como o exemplo seguinte que define um método *static* que coloca referências de um único item em várias caixas (`Boxes`):

```
1. public static <U> void fillBoxes(U u, List<Box<U>> boxes) {
2.    for (Box<U> box : boxes) {
3.       box.add(u);
4.    }
5. }
```

Figura 14 - Método genérico `fillBoxes`

Para usar este método o código deverá ser algo como:

```
1. Pencil red = ...;
2. List<Box<Pencil>> pencilBoxes = ...;
```

A sintaxe completa para invocar o método é:

```
1. Box.<Pencil>fillBoxes(red, pencilBoxes);
```

Aqui, fornecemos explicitamente o tipo a ser usado como `U`, mas na maioria das vezes isso pode ser deixado de fora e o compilador inferirá o tipo necessário:

```
1. Box.fillBoxes(red, pencilBoxes); // compiler infers that U is Pencil
```

Este recurso, conhecido como inferência de tipo, permite invocar um método genérico como faria com um método normal sem especificar um tipo entre <>.

2.5. Genéricos Limitados

Os tipos parametrizados também podem ser limitados (restritos) ao restringir os tipos que um método aceita. Por exemplo, podemos especificar que um método aceita um tipo e todas as suas subclasses (limite superior) ou um tipo e todas as suas superclasses (limite inferior). Para declarar um tipo de limite superior, usamos a palavra reservada `extends` após o tipo, seguida pelo limite superior que queremos usar:

```
1. public <T extends Number> List<T> fromArrayToList(T[] a) {
2.     ...
3. }
```

Figura 15 - Declaração de um tipo de limite superior

A utilização da palavra reservada `extends` aqui significa que o tipo `T` estende o limite superior no caso de uma classe, ou implementa um limite superior no caso de uma interface.

2.6. Vários Limites

Um tipo também pode ter vários limites superiores:

```
1. <T extends Number & Comparable>
```

Se um dos tipos estendidos por `T` for uma classe (por exemplo, `Number`), temos que colocá-lo primeiro na lista de limites. Caso contrário, irá causar um erro após a compilação.

2.7. Wildcards

Um *wildcard* em Java é um conceito útil em Java Generics. Existem 3 tipos de *wildcards* em Java: *wildcards* de limite superior, *wildcards* de limite inferior e *wildcards* ilimitados.

Os *wildcards* não são nada mais que o ponto de interrogação (?) que usamos com os genéricos. Podemos usar *wildcards* como variável, parâmetro ou como tipo de retorno. No entanto, quando a classe genérica é instanciada ou quando um método genérico é invocado, não podemos usar *wildcards*. Usamos amplamente Java *wildcards* em situações como um tipo parametrizado, variável local e também como um tipo de retorno. Ao contrário dos arrays as diferentes instanciações de um tipo genérico não são compatíveis entre si, nem mesmo explicitamente.

Os *wildcards* são úteis para remover a incompatibilidade entre diferentes instanciações de um tipo genérico. Essa incompatibilidade é removida usando o *wildcard* ? como um parâmetro de tipo real.

Wildcards de limite superior

Os *wildcards* de limite superior são os *wildcards* que relaxam a restrição do tipo de variável. No exemplo apresentado na figura 16 é usado um *wildcard* de limite superior no método `sum` que calcula a soma de qualquer tipo de variável, seja do tipo `int` ou `double`.

```
1. public class UpperBoundWildcard {
2.   public static void main(String[] args) {
3.     //Upper Bounded Integer List
4.     List < Integer > intList = Arrays.asList(10, 20, 30, 40);
5.
6.     //Printing the sum of integer elements in list
7.     System.out.println("Total sum is:" + sum(intList));
8.
9.     //Upper Bounded Double list
10.    List < Double > doubleList = Arrays.asList(13.2, 15.6, 9.7, 22.5);
11.
12.    //Printing the sum of double elements in list
```

```
13.     System.out.print("Total sum is: " + sum(doubleList));
14.   }
15.
16.   private static double sum(List <? extends Number > myList) {
17.     double sum = 0.0;
18.     for (Number iterator: myList) {
19.       sum = sum + iterator.doubleValue();
20.     }
21.     return sum;
22.   }
23.}
```

Figura 16 - *Wildcard* de limite superior no método `sum`

***Wildcards* de limite inferior**

Os *wildcards* de limite inferior são usados para ampliar o uso do tipo de variável. Por exemplo, se pretendermos adicionar uma lista de inteiros a um determinado método, podemos usar o `List<Integer>` no entanto estamos a obrigar a que possa apenas ser usada uma lista de inteiros.

Para usarmos uma `List<Number>` ou uma `List<Object>` para armazenar a lista de inteiros temos de usar um *wildcard* de limite inferior. Para isso usamos o *wildcard* `?` juntamente com a palavra reservada `super` seguida do limite inferior: `<? super LimiteInferior>`.

```
1. import java.util.*;
2. public class LowerBoundWildcard {
3.     public static void main(String[] args) {
4.        //Lower Bounded Integer List
5.        List < Integer > intList = Arrays.asList(10, 20, 30, 40);
6.
7.        //Passing Integer list object
8.        printOnlyIntegerClassorSuperClass(intList);
9.
10.       //Number list
11.       List < Number > numberList = Arrays.asList(10, 20, 30, 40);
```

```
12.
13.       //Passing Integer list object
14.       printOnlyIntegerClassorSuperClass(numberList);
15.    }
16.
17.    public static void printOnlyIntegerClassorSuperClass(
18.                                       List <? super Integer > list) {
19.       System.out.println(list);
20.    }
21.}
```

Figura 17 - Exemplo de *wildcard* de limite inferior

***Wildcards* ilimitados**

Wildcards ilimitados são usados com recurso ao tipo de *Wildcard* com o caractere `?`. Geralmente usamos esse *Wildcard* quando o código dentro do método está a usar a funcionalidade de `Object` e também quando o código dentro do método não depende do tipo parametrizado.

```
1. import java.util.*;
2. public class UnboundedWildcard {
3.    public static void main(String[] args) {
4.       //Integer List
5.       List < Integer > intList = Arrays.asList(10, 20, 30, 40);
6.
7.       //Double list
8.       List < Double > doubleList =
9.                         Arrays.asList(11.5, 13.6, 67.8, 43.7);
10.
11.      printList(intList);
12.      printList(doubleList);
13.   }
14.
15.   private static void printList(List <?>list) {
16.    System.out.println(list);
17.   }
```

```
18. }
```

Figura 18 - Exemplo de *wildcard* ilimitado

2.8. Eliminação de Tipo

Os genéricos foram adicionados ao Java para garantir a segurança de tipo. Para garantir que os genéricos não causam qualquer sobrecarga em tempo de execução o compilador aplica um processo chamado eliminação de tipo em genéricos em tempo de compilação.

A eliminação de tipo remove todos os tipos parametrizados e substitui-os pelos seus limites ou por `Object` se o tipo parametrizado for ilimitado. Desta forma, o *bytecode* após o processo de compilação contém apenas classes, interfaces e métodos normais, garantindo que nenhum tipo novo será produzido. A conversão adequada também é aplicada ao tipo de objeto em tempo de compilação.

Este é um exemplo de eliminação de tipo:

```
1. public <T> List<T> genericMethod(List<T> list) {
2.     return list.stream().collect(Collectors.toList());
3. }
```

Figura 19 - Exemplo de eliminação de tipo com tipo genérico

Com a eliminação de tipo, o tipo ilimitado `T` é substituído por `Object`:

```
1. // For illustration
2. public List<Object> withErasure(List<Object> list) {
3.     return list.stream().collect(Collectors.toList());
4. }
5.
6. // Which in practice results in
7. public List withErasure(List list) {
8.     return list.stream().collect(Collectors.toList());
```

```
9. }
```

Figura 20 - Exemplo de eliminação de tipo

Se o tipo for limitado, o tipo será substituído pelo limite em tempo de compilação:

```
1. public <T extends Building> void genericMethod(T t) {
2.     ...
3. }
```

Irá mudar após a compilação:

```
1. public void genericMethod(Building t) {
2.     ...
3. }
```

2.9. Genéricos e Tipos Primitivos

Uma restrição dos genéricos em Java é que o tipo parametrizado não pode ser um tipo primitivo.

Por exemplo, o código apresentado de seguida não compila:

```
1. List<int> list = new ArrayList<>();
2. list.add(17);
```

Para compreender porquê que os tipos de dados primitivos não funcionam temos que nos lembrar que os genéricos são um recurso em tempo de compilação, o que significa que o tipo parametrizado é eliminado e todos os tipos genéricos são implementados como tipo `Object`.

Vejamos o método `add` de uma lista:

```
1. List<Integer> list = new ArrayList<>(); list.add(17);
```

A assinatura do método `add` é:

```
1. boolean add(E e);
```

que será compilado para:

```
1. boolean add(Object e);
```

Portanto, os tipos parametrizados devem ser "possíveis de converter" para `Object`. Como os tipos primitivos não estendem `Object` não podemos usá-los como tipos parametrizados.

No entanto, Java fornece "*boxed types*", juntamente com *autoboxing* e *unboxing* para auxiliar neste processo:

```
1. Integer a = 17;
2. int b = a;
```

Então, se quisermos criar uma lista que possa conter inteiros podemos usar este *wrapper*:

```
1. List<Integer> list = new ArrayList<>();
2. list.add(17);
3. int first = list.get(0);
```

O código compilado será o equivalente ao seguinte:

```
1. List list = new ArrayList<>();
2. list.add(Integer.valueOf(17));
3. int first = ((Integer) list.get(0)).intValue();
```

2.10. Exercícios Propostos

Exercício 1

Implemente uma classe `Par` que possa armazenar dois objetos declarados como tipos genéricos. Demonstrar o uso da classe `Par` numa classe `Demo` que possua o método `main` e que permita criar e imprimir objetos do tipo `Par` com cinco tipos diferentes de pares, como, por exemplo:

- `<String,Double>` (nome e nota de um aluno)
- `<Integer,String>` (código e nome de um funcionário)
- `<Float,Float>` (coordenadas x e y)

Exercício 2

Imagine que uma empresa codifica os seus produtos em 2 partes: as letras indicam o setor onde o produto é fabricado e os números são um código sequencial dentro do setor

- Ex: "IMP34", "SEC1413", ...
- Criar uma classe denominada `Codigo` que represente esses códigos e instancie a classe `ProdutoG` que use a classe `Codigo` como tipo do código do produto.

```
1. public class ProdutoG2<T,U> {
2.     private T codigo;
3.     private String descricao;
4.     private U preco;
5.
6.     public ProdutoG2(T cod, String descr, U pr) {
7.          codigo = cod;
8.          descricao = descr;
9.          preco = pr;
10.     }
11.
12.     public T getCodigo() { return codigo; }
13.     public String getDescricao() { return descricao; }
14.     public U getPreco() { return preco; }
15.
16.     @Override
17.     public String toString() {
```

```
18.         return "ProdutoG2 {" + "codigo=" + codigo + ", descricao=" +
19.         descricao + ", preco=" + preco + "}";
20.     }
21.}
```

Imagine que os tipos dos componentes da classe `Codigo` podem variar: altere a classe `Codigo` para que use dois tipos parametrizados (para a primeira e a segunda parte do código) e use esta nova classe para instanciar a classe `ProdutoG` .

Exercício 3

Criar uma classe `ValorM` que represente um valor monetário, e use-a juntamente com a classe `Codigo` do **exercício 2** para instanciar a classe `ProdutoG` vista anteriormente.

Exercício 4

Um dicionário é uma estrutura onde os elementos são armazenados sob uma chave. Por exemplo, podemos armazenar os dados sobre um automóvel através da chave matrícula. Para recuperar os dados do automóvel basta fornecer a matrícula (ao invés, por exemplo, da posição onde os dados foram armazenados)

Através da utilização de dois *arrays*, crie uma classe `Dicionario` onde tanto a chave como o valor têm o seu tipo definido por tipos parametrizados

Imagine que as operações seguintes devem estar disponíveis:

- `add(K chave, V valor)` - adicionar à chave K o valor V
- `V get(K chave)` - retorna o valor associado à chave K (se não encontrar, retorna null)

Nota: para criar um *array* de genéricos, use:

- `E[] vet = (E[]) new Object[TAMANHO];`