Métodos genéricos

Método para ver se duas listas são iguais – 1ª tentativa:

```
static boolean iguais(List<Object> a, List<Object> b) {
     boolean result = a.size() == b.size();
     for (int i = 0 ; i < a.size() && result ; i++)
          if(!a.get(i).equals(b.get(i)))
              result = false;
     return result;
}
```

Não posso instanciar a nem b com ArrayList<Number> p. ex

Método para ver se duas listas são iguais – 2ª tentativa:

```
static boolean iguais(List<?> a, List<?> b) {
     boolean result = a.size() == b.size();
     for (int i = 0 ; i < a.size() && result ; i++)
          if(!a.get(i).equals(b.get(i)))
              result = false;
     return result;
}
```

Posso invocar com listas de elementos de tipos diferentes... Como obrigar a que seja o mesmo?

Métodos genéricos

Método para ver se duas listas são iguais – 3ª tentativa:

```
static <T> boolean iguais(List<T> a, List<T> b) {
     boolean result = a.size() == b.size();
     for (int i = 0 ; i < a.size() && result ; i++)
          if(!a.get(i).equals(b.get(i)))
              result = false;
     return result;
}
```

Este método é genérico pois tem um parâmetro de tipo.

Podemos chamar este método com 2 listas de qualquer tipo desde que os elementos da 2ª lista sejam exatamente do mesmo tipo que os elementos da 1ª lista.

Ao invocá-lo, não temos que explicitar o tipo que instancia T. O compilador infere isso por nós.

Métodos genéricos

Método genérico para ver se os elementos de uma lista são o dobro dos de outra:

```
static <T extends Number> boolean dobro(List<T> a, List<T> b) {
        boolean result = a.size() == b.size();
        for (int i = 0 ; i < a.size() && result ; i++){
            double ad = a.get(i).doubleValue();
            double bd = b.get(i).doubleValue();
             if(!(Math.abs(ad - 2 * bd) < 0.0001))
                 result = false;
        }
        return result;

}
```

Aqui o parâmetro de tipo é ***bounded*** pois queremos restringir o tipo dos elementos das listas parâmetro – têm que ser números.

Métodos genéricos

Exemplos de invocação:

```
List<Object> lo1 = new ArrayList<Object>();
List<Object> lo2 = new ArrayList<Object>();
boolean b = iguais(lo1,lo2);  // T inferred to be Object

List<String> ls1 = new ArrayList<String>();
List<String> ls2 = new ArrayList<String>();
b = iguais(ls1,ls2);          // T inferred to be String

b = iguais(ls1,lo1);          // compile-time error

List<Number> ln1 = new ArrayList<Number>();
List<Integer> li1 = new ArrayList<Integer>();
b = iguais(li1,ln1);          // compile-time error

List<Number> ln2 = new ArrayList<Number>();
ln1.add(2); ln1.add(3);
ln2.add(1); ln2.add(6);
b = dobro(ln1,ln2);          // T inferred to be Number

List<Integer> li2 = new ArrayList<Integer>();
li1.add(64); li1.add(18);
li2.add(32); li2.add(9);
b = dobro(li1,li2);          // T inferred to be Integer
```

Métodos genéricos

Quando usar métodos genéricos?

Na interface Collection temos, por exemplo:

```
interface Collection<E> {
     public boolean containsAll(Collection<?> c);
     public boolean addAll(Collection<? extends E> c);
}
```

Podíamos ter, no entanto:

```
interface Collection<E> {
     public <T> boolean containsAll(Collection<T> c);
     public <T extends E> boolean addAll(Collection<T> c);
}
```

Mas preferimos usar *wildcards* pois nestes casos o parâmetro de tipo só é usado uma vez em cada método.

Métodos genéricos

Precisamos de usar um parâmetro de tipo para obrigar a que um tipo referido em dois parâmetros, ou num parâmetro e no tipo de retorno, são o mesmo:

Na classe Collections temos, por exemplo:

```
class Collections {
   public static <T> void copy(List<? super T> d, List<? extends T> s) {
         ...
     }
     ...
}
```

Aqui temos dependência entre os tipos dos 2 parâmetros. Já devemos usar um método genérico!

Alguns métodos da classe Collections

- **Classe Collections** – Contém métodos de classe (static) que executam funções úteis sobre as colecções. Exemplos:

java.util
Class Collections

java.lang.Object
  └ java.util.Collections

Shuffling

static void shuffle(List<?> list)

Sorting

static <T> void sort(List<T> list, Comparator<? Super T> comp)

static <T extends Comparable<? super T>> void sort(List<T> list)

Searching

static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)

Finding Extreme Values

... min(Collection<? extends T> col)

.... max(Collection<? extends T> col)