Generics e Collections

Prof. Dr. Lucas C. Ribas

Disciplina: Programação Orientada a Objetos

Departamento de Ciências de Computação e Estatística

unesp
UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"

IBILCE / UNESP - CÂMPUS DE SÃO JOSÉ DO RIO PRETO

Agenda

- *Generics*
- Classes e métodos genéricos
- Genéricos restritos
- Herança de classes genéricas
- *Wildcards*
- Collections
- Interfaces e classes de coleções
- Algoritmos sobre genéricos
- Iteradores

unesp

Generics

Introdução

- Generics, em Java, é um recurso semelhante às templates de C+
- Generics permite que tipos (classes e interfaces) sejam parâmetros na definição de classes, interfaces e métodos
- Permite o reuso de código para diferentes tipos
  - Sobrecarga de métodos, em geral, replica o mesmo código para tipos de dados diferentes
  - Evita o uso de casting explícito por parte do programador

unesp

Tipos Genéricos

- Um **tipo genérico** é uma classe ou interface parametrizada sobre tipos
- Considere a classe abaixo, que não usa Generics
  - Qualquer objeto pode ser armazenado na caixa
  - Em tempo de compilação, não é possível saber que tipo de objeto será passado (set) ou retornado (get)

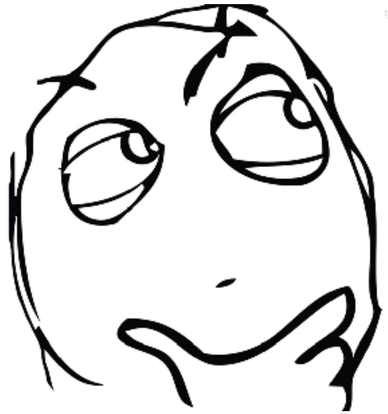
```
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

unesp

Tipos Genéricos

- A versão abaixo utiliza *Generics*:
  - Parâmetro é incluído entre colchetes angulares <> (diamond) logo após o nome da classe
  - O tipo **T** pode ser qualquer tipo não primitivo



```
public class Box<T> { // T stands for "Type“
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

Contenção de Nomes

- Qualquer palavra não-chave pode ser usada para especificar um parâmetro de tipo
- Contudo, por convenção, tipos são definidos com uma única letra maiúscula
  - E-Element(JavaCollections)
  - K-Key
  - N-Number
  - T-Type
  - V-Value
  - S,U,V etc.-2nd, 3rd, 4th types

unesp

Declaração e Instanciação

- A declaração e a instanciação de um tipo genérico deve especificar qual o tipo desejado
- Similar à chamada de um método ou construtor, para o qual passamos parâmetros
  - Porém, em Generics, o parâmetro é um tipo (classe ou interface)

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

```
Box<Integer> integerBox;
integerBox = new Box<>(); // from Java SE 7
```

unesp

Declaração e Instanciação

- Tipos genéricos não podem receber tipos primitivos, como comentado anteriormente
- Para estes casos, Java possui classes que representam os tipos primitivos
  - int → Integer
  - double → Double
  - …
- A conversão é feita automaticamente pelo Java

```
Box<Integer> integerBox = new Box<Integer>();
integerBox.set(10);
```

unesp

Múltiplos Tipos

- Uma classe genérica pode ter múltiplos parâmetros de tipo

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}
```

```
public class OrderedPair<K, V> implements Pair<K, V> {
    private K key;
    private V value;

    public OrderedPair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() { return key; }
    public V getValue() { return value; }
}
```

unesp

Múltiplos Tipos

- Com a definição da interface Pair anterior, existem inúmeras possibilidades de criar pares chave-valor

```
Pair<String, Integer> p1;
Pair<String, String> p2;

p1 = new OrderedPair<String, Integer>("Even", 8);
p2 = new OrderedPair<String, String>("hello", "world");
```

```
Pair<String, Integer> p1 = new OrderedPair<>("Even", 8);
Pair<String, String> p2 = new OrderedPair<>("hello", "world");
```

```
OrderedPair<String, Box<Integer>> p;
p = new OrderedPair<>("primes", new Box<Integer>(...));
```

unesp

Métodos Genéricos

- É possível aplicar o conceito de Generics apenas em métodos, sem que a classe como um todo seja genérica
  - Métodos estáticos e não estáticos, construtores
- A lista dos parâmetros de tipo deve aparecer antes do tipo de retorno

```
public class PairUtil {
    public static <K, V> boolean compare(Pair<K, V> p1,
                                         Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
               p1.getValue().equals(p2.getValue());
    }
}
```

unesp

Métodos Genéricos

- Usando o método genérico

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = PairUtil.<Integer, String>compare(p1, p2);
```

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = PairUtil.compare(p1, p2);
```

unesp

Genéricos Restritos

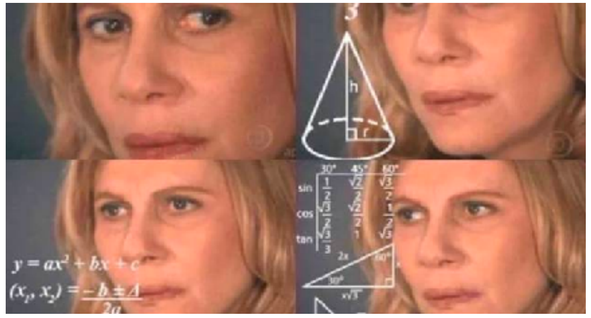
- As vezes queremos restringir os tipos que podem ser utilizados em uma classe genérica
  - Por exemplo, uma caixa que guarda apenas números
- É possível forçar que o tipo genérico herde uma determinada classe ou implemente uma interface
  - Uso da palavra chave extends(mesmo para interfaces)
  - Limite superior

```
public class Box<T extends Number> {
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

unesp

Genéricos Restritos



- Qual a diferença entre as declarações abaixo?
  - Quem chamar **get()**, receberá tipos diferentes

```
public class Box<T extends Number> {
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

```
public class Box {
    private Number t;

    public void set(Number t) { this.t = t; }
    public Number get() { return t; }
}
```

Genéricos Restritos

- Outro exemplo usando a interface **Comparable**

```
public static <T extends Comparable<T>> int countGreaterThan(
                                                T[] anArray, T elem) {
    int count = 0;

    for (T e : anArray)
        if (e.compareTo(elem) > 0)
            ++count;

    return count;
}
```

unesp

Múltiplas Restrições

- É possível que o tipo genérico seja restrito não apenas por uma classe ou interface, mas sim por várias

```
public class D <T extends A & B & C> {
    // code
}
```

- Se houver uma classe entre os tipos de restrição (A, B ou C), é preciso declarar a classe em primeiro lugar
  - Neste caso, **A** é uma classe enquanto **B** e **C** são interfaces

unesp

Herança de Classes Genéricas

- Quando criamos uma caixa de números, qualquer tipo numérico pode ser armazenado na caixa

```
Box<Number> box = new Box<Number>();
box.add(new Integer(10)); // OK
box.add(new Double(10.1)); // OK
```

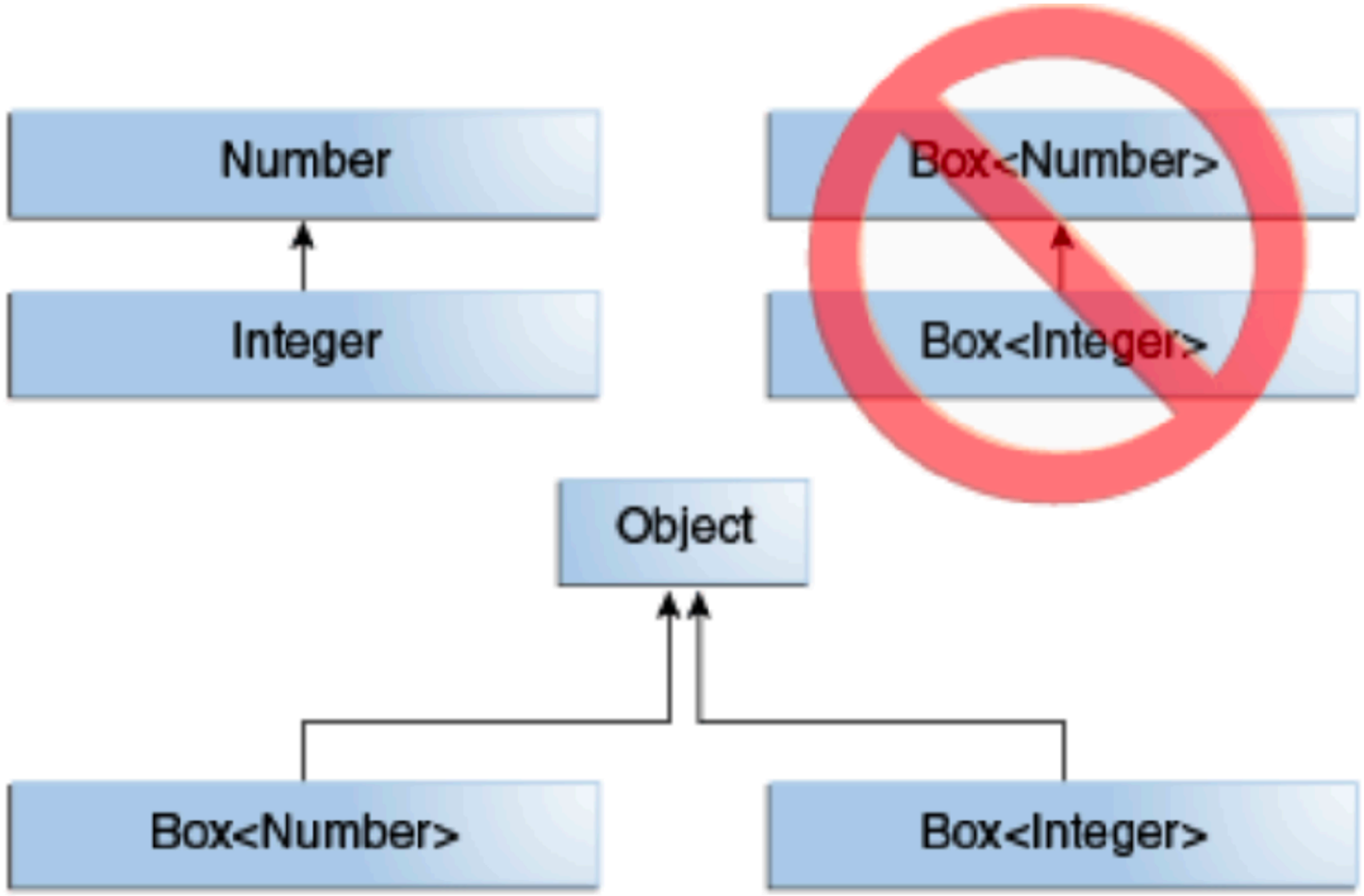
- Porém, no método abaixo

```
public void boxTest(Box<Number> n) { /* ... */ }
```

- não podemos passar Box<Integer> ou Box<Double>
  - Não há relação de herança entre essas classes

unesp

Herança de Classes Genéricas

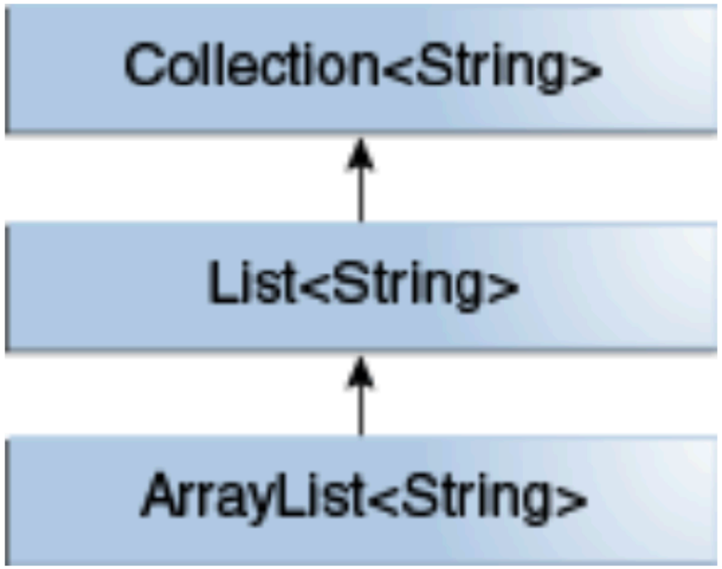


Herança de Classes Genéricas

- Podemos herdar ou implementar classes genéricas
  - A classe filha pode ser genérica ou não
  - A relação entre os tipos genéricos da classe pai e filha é definida na declaração da classe
- Exemplo
  - **ArrayList<E>** implementa **List<E>**
  - **List<E>** herda de **Collection<E>**
  - Importante: note, porém, que a relação de herança vale apenas quando os tipos são iguais

unesp

Herança de Classes Genéricas



Herança de Classes Genéricas

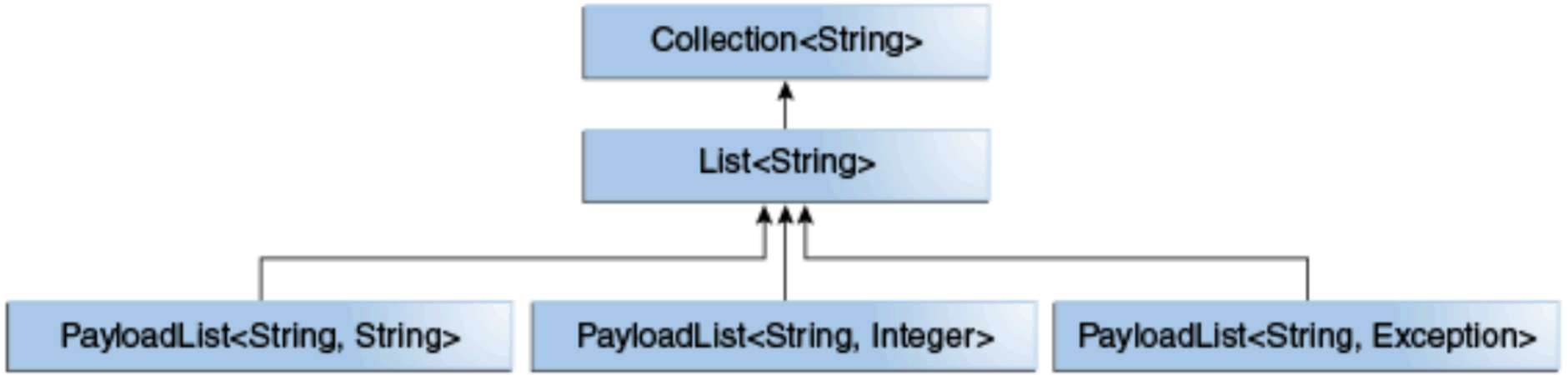
- Considere uma nova interface que herda de List

```
interface PayloadList<E,P> extends List<E> {
    void setPayload(int index, P val);
    ...
}
```

- Neste caso, várias parametrizações de PayLoadList serão filhas de List<String>
  - PayLoadList<**String**,String>
  - PayLoadList<**String**,Integer>
  - PayLoadList<**String**,Exception>
  - ...

unesp

Herança de Classes Genéricas



Wildcards

Wildcards

- Como já vimos, o método abaixo recebe uma caixa contendo um elemento do tipo Number

```
public void boxTest(Box<Number> n)
```

  - Porém, o método suporta caixas apenas com tipo Number (não adianta ser filho de Number)

- Para diminuir a restrição do tipo, podemos usar o wildcard (curinga)

  - Ponto de interrogação (?)

```
public void boxTest(Box<?> n)
```

Wildcards

- No exemplo abaixo, apenas lista de Objetos são aceitos pelo método

```
public static void printList(List<Object> list) {
    for (Object elem : list)
        System.out.println(elem + " ");
    System.out.println();
}
```

- O método abaixo serve para listas de qualquer tipo

```
public static void printList(List<?> list) {
    for (Object elem : list)
        System.out.println(elem + " ");
    System.out.println();
}
```

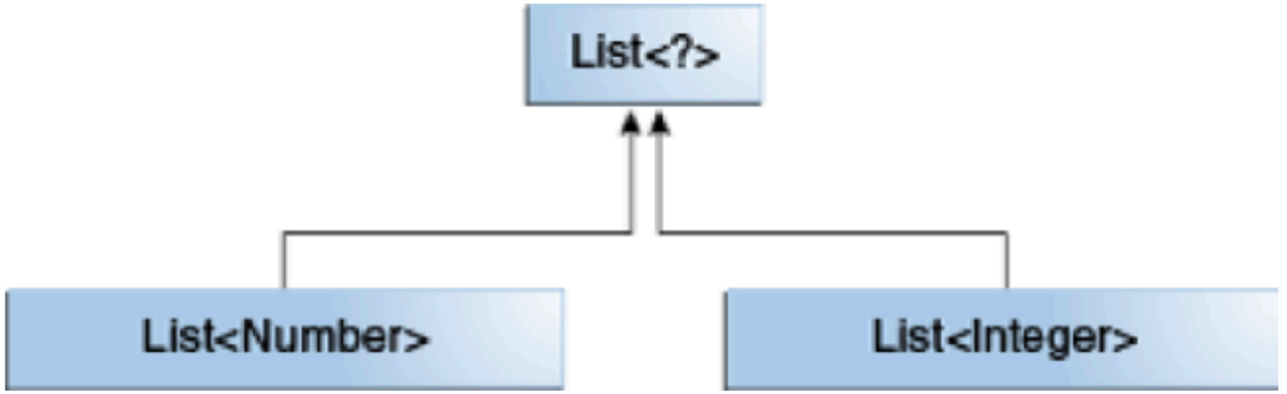
Wildcards

- Para qualquer tipo concreto A, List<A> é filha de List<?>
- Sendo assim, List<?> aceita listas de qualquer tipo
  - List<Object>
  - List<String>
  - ...

```
List<Integer> li = Arrays.asList(1, 2, 3);
List<String> ls = Arrays.asList("one", "two", "three");
printList(li);
printList(ls);
```

unesp

Wildcards



Wildcards

- Vimos que o wildcard permite que uma classe genérica represente vários tipos

```
public void boxTest(Box<Number> n)
```

```
public void boxTest(Box<?> n)
```

- Qual a diferença entre os dois métodos acima?
  - O primeiro recebe apenas caixa do tipo Number
  - O segundo recebe caixas de qualquer tipo
- Podemos adicionar um limite superior a um wildcard, um meio termo entre as duas declarações acima

```
public void boxTest(Box<? extends Number> n)
```

Wildcards

```
public void boxTest(Box<? extends Number> n)
```

- A declaração acima serve para caixa do tipo Number e todos os subtipos de Number
- Isso quer dizer que Box<? extends Number> é pai de
  - Box<Integer>
  - Box<Double>
  - Box<Float>
  - ...

Wildcards

- De maneira similar ao limite superior, é possível definir um limite inferior para *wildcards*:
- Limite superior -> **extends**:
  - Neste caso, os tipos devem ser filhos da classe especificada após **extends**
- Limite inferior -> **super**:
  - Neste caso, o tipo deve ser pai da classe especificada após **super**
- O exemplo abaixo suporta caixas de **Integer**, **Number** e **Object**

```
public void boxTest(Box<? super Integer> n)
```

unesp

Relembrando...

- É possível restringir
  - extends ou super

```
public class Box<T> {
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

- Podemos parametrizar apenas um método

```
public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
```

- Cuidado quando uma classe genérica é parâmetro de um método

```
public void boxTest(Box<Number> n) {
```

  - Box<Number> não é pai de Box<Integer> ou Box<Double>
  - Devemos usar wildcard se quisermos permitir esses tipos

```
public void boxTest(Box<?> n) {
```

Wildcards

- As duas definições abaixo produzem o mesmo efeito

```
public void boxTest(Box<? extends Number> n)
```

```
public <T extends Number> void boxTest(Box<T> n)
```

- Existe alguma implicação em usar uma ou outra?
- Quando usamos *wildcards*, não é possível referenciar o tipo genérico
  - **?** não é um tipo como **T**
  - **?** representa qualquer tipo

unesp

Wildcards

```
static void fromArrayToCollection(Object[] a, Collection<?> c) {
    for (Object o : a)
        c.add(o); // compile-time error
}
```

```
static <T> void fromArrayToCollection(T[] a, Collection<T> c) {
    for (T o : a)
        c.add(o); // Correct
}
```

- No primeiro caso, como o tipo dos elementos da coleção é desconhecido, é seguro lê-los como **Object**
- Porém, não se pode adicionar nada
  - Não há garantia de que a coleção é de Object

Wildcards

- Assim, quando devemos usar **métodos genéricos** e quando devemos usar **wildcards**?
  - Se o intuito é apenas flexibilizar um parâmetro, use wildcards
    - **T** aparece apenas um vez
  - Use método genérico quando há uma relação de dependência entre os tipos dos parâmetros, tipo de retorno do método, etc.
    - **T** aparece em vários lugares

```
public boolean containsAll(Collection<?> c);  // better
public <T> boolean containsAll(Collection<T> c);
```

```
public static <T> void copy(List<T> dest, List<? extends T> src);
public static <T, S extends T> void copy(List<T> dest, List<S> src);
// S is not necessary
```

unesp

Apagamento (*erasure*) de Tipos

- O compilador Java substitui os tipos genéricos por tipos concretos na definição das classes e métodos
  - Utiliza o limite superior da definição do tipo (extends)
  - Senão houver, assume **Object**
- Perceba que isso, por si só, não oferece a funcionalidade esperada
  - O compilador também adiciona castings explícitos em cada chamada de métodos, de acordo com tipos definidos
  - É possível que o compilador crie bridge methods para preservar o polimorfismo
    - https://docs.oracle.com/javase/tutorial/java/generics/bridgeMethods.html

unesp

Limitações de *Generics*

- Não é possível instanciar um tipo genérico usando tipos primitivos

```
Pair<int, char> p = new Pair<>(8, 'a'); // compile-time error
Pair<Integer, Character> p = new Pair<>(8, 'a'); // OK
```

- Não é possível criar instâncias de parâmetros genéricos

```
public static <E> void append(List<E> list) {
    E elem = new E(); // compile-time error
    list.add(elem);
}
```

unesp

Limitações de *Generics*

- Não é possível declarar campos estáticos de tipos genéricos
  - Um campo estático é compartilhado por todas as instâncias da classe, o que geraria inconsistência de tipos

```
public class MobileDevice<T> {
    private static T os;
    // ...
}
```

```
MobileDevice<Smartphone> phone = new MobileDevice<>();
MobileDevice<Pager> pager = new MobileDevice<>();
MobileDevice<TabletPC> pc = new MobileDevice<>();
```

unesp

Limitações de *Generics*

- Não é possível criar arrays de tipos parametrizados

```
// compile-time error
List<Integer>[] arrayOfLists = new List<Integer>[2];
```

- Não é possível criar objetos de exceção (que herdam Throwable direta ou indiretamente) com tipo parametrizado
  - Logo, também não é possível capturar (**catch**) ou lançar (**throw**) objetos com tipos parametrizado
  - Porém, podemos parametrizar a cláusula throws de um método

```
public <T extends Exception> void parse(File file) throws T {
```

unesp

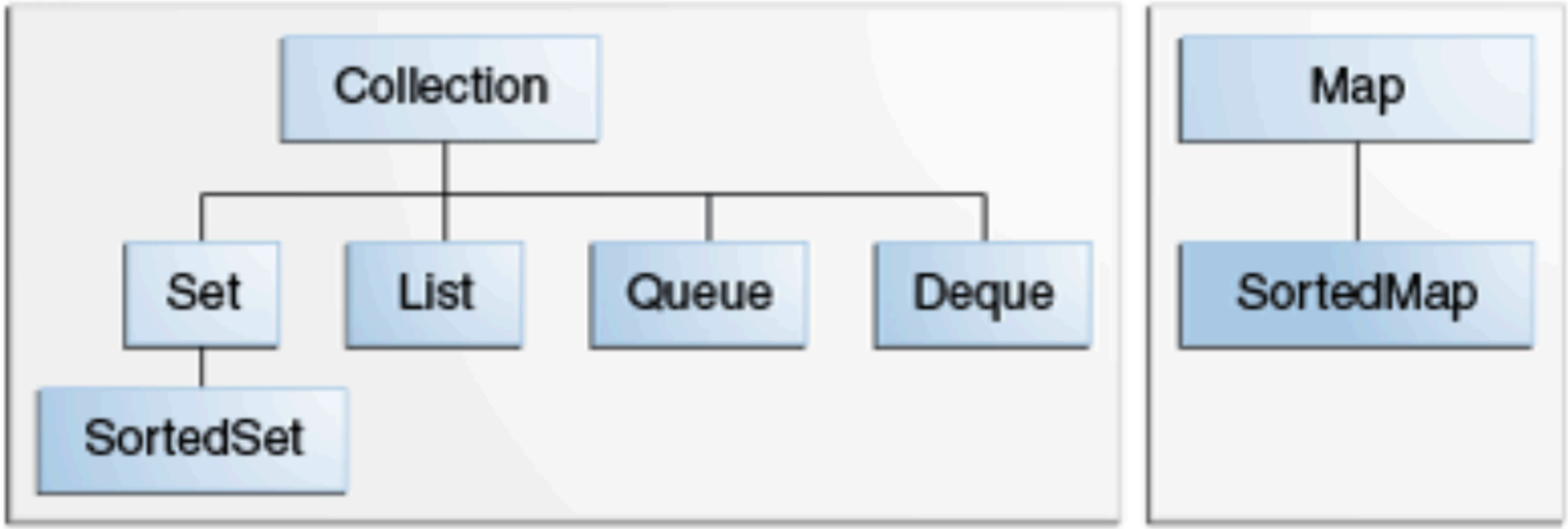
Collections

Collections

- Uma coleção em Java nada mais é do que um objeto que agrupa vários elementos
- O framework collections da API Java oferece:
  - **interfaces**: definições abstratas de coleções
  - **Implementações**: objetos concretos de coleções
  - **Algoritmos**: vários métodos para trabalhar com coleções
    - por exemplo, busca e ordenação.
- O framework **collections** utiliza *generics*

unesp

Interfaces Collections



Interfaces Collections

- Collection:
  - É a raiz da hierarquia de coleções
  - Não determina nenhuma restrição para coleções (ordem, duplicação de elementos, etc)
  - Na API, nenhuma classe concreta implementa diretamente esta interface

Interfaces Collections

- Set:
  - Coleção que **não permite elementos duplicados**
  - Representa a definição de conjuntos em Matemática
- SortedSet:
  - Igual a Set
  - Porém, mantém os elementos ordenados (ascendente)
- List:
  - Coleção **ordenada** de elementos (sequências)
  - Pode conter elementos duplicados
  - Há controle preciso sobre a posição de cada elemento (indexação)

unesp

Interfaces Collections

- Queue:
  - Coleção que representa uma fila genérica
  - Em geral, utiliza a regra FIFO (*First In, First Out*)
- Deque:
  - Semelhante a Queue
  - Porém, inserções e remoções podem ser feitas em qualquer extremidade
  - Também pode ser usada para representar uma pilha (LIFO - *Last In, First Out*)

unesp

Interfaces Collections

- Map:
  - Coleção de pares chave-valor
  - Cada chave mapeia apenas um valor
  - Chaves não podem ser duplicadas
  - Representa a definição de função em Matemática
- SortedMap:
  - Igual a Map
  - Porém, mantém os elementos ordenados pela chave, em ordem ascendente
  - Exemplos: dicionário, lista telefônica.

unesp

Classes Concretas Collections

- **Set**
  - HashSet
    - Armazena os elementos em uma tabela hash
    - Melhor performance
    - Não garante nenhum tipo de ordem dos elementos
  - TreeSet
    - Usa uma estrutura de árvore (vermelho-preta) para armazenar os elementos
    - Mantém a ordem dos elementos
    - Bem mais lenta que a **HashSet**
  - LinkedHashSet
    - Usa tabela hash com lista ligada
    - Mantém a ordem de inserção dos elementos
    - Um pouco menos eficiente que HashSet

unesp

Classes Concretas Collections

- **List**
  - ArrayList
    - Implementação diretamente indexada de elementos (arrays)
    - De maneira geral, implementação mais eficiente
    - Requer a realocação de memória quando a lista precisa crescer além do tamanho inicialmente alocado
    - Não é muito eficiente em operações que envolvem a inserção ou remoção de elementos no meio da lista
  - LinkedList
    - Usa lista duplamente encadeada para armazenar os elementos
    - Acesso aleatório (indexado) exige percorrer a lista
    - É adequado para listas que precisam ser modificadas com frequência,
    - Não exige realocação de memória quando a lista cresce ou diminui

unesp

Classes Concretas Collections

- **Queue**:
  - **add** e **offer** inserem um elemento na fila
  - **remove** e **poll** removem e retornam o elemento do início
  - **element** e **peek** retornam, mas não removem, o elemento do início
  - LinkedList
  - PriorityQueue:
    - Prioridade é determinada pelo próprio valor do elemento.
    - Elementos precisam ser do tipo Comparable.

unesp

Classes Concretas Collections

- **Deque**
  - Inserção
    - addFirst, offerFirst, addLast, offerLast
  - Remoção
    - removeFirst, poolFirst, removeLast, poolLast
  - ArrayDeque: array
  - LinkedList: lista ligada
  - Note que **LinkedList** implementa as interfaces List, Queue e Deque

unesp

Classes Concretas Collections

- **Map**
  - HashMap: tabela hash
  - TreeMap: árvore
  - LinkedHashMap: tabela hash e lista ligada
  - Comportamento e performance similar ao visto para Set
  - Por armazenar um par chave-valor, há dois parâmetros genéricos

```
// Map<K, V>
Map<String, Integer> m = new HashMap<String, Integer>();
...
```

unesp

Algoritmos sobre Collections

- A classe Collections provê alguns métodos estáticos para manipular coleções
- Ordenação
  - <T extends Comparable<? super T>> void **sort**(List<T>)
    - Note que T precisa ser do tipo **Comparable**
  - <T> void **sort**(List<T>, Comparator<? super T>)
    - Usa o objeto **Comparator** passado para ordenar
    - Permite ordenar utilizando um critério diferente do que foi definido por ordem natural (**Comparable**)
  - Utiliza uma versão otimizada do *merge sort*
    - Garante performance **n log n** e **estabilidade**

unesp

Algoritmos sobre Collections

- <T extends Comparable<? super T>> void **sort**(List<T>)

```
import java.util.*;
class Person implements Comparable<Person> {
   String name;

   Person(String name) {
        this.name = name;
   }
   public int compareTo(Person other) {
        return this.name.compareTo(other.name);
    }
   public String toString() {
        return this.name;
    }
}
public class Main {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        people.add(new Person("Carlos"));
        people.add(new Person("Ana"));
        people.add(new Person("Bruno"));

        System.out.println("Before sorting: " + people);

        Collections.sort(people);

        System.out.println("After sorting: " + people);
    }
}
```

Algoritmos sobre Collections

- Embaralhamento (*shuffling*)
  - void **shuffle**(List<T>)
  - void **shuffle**(List<T>, Random)
  - Collections.shuffle(lista);
- Métodos de propósito geral:
  - **reverse**: inverte os elementos de uma lista
  - **fill**: substitui todos os elementos de uma lista
  - **copy**: copia os elementos de uma lista para outra lista (sobrescreve)
  - **swap**: troca a posição de dois elementos de uma lista
  - **addAll**: adiciona elementos em lote em uma coleção

unesp

Algoritmos sobre Collections

- Busca binária:
  - <T> int **binarySearch**(List<? extends Comparable<? super T>> list, T key)
  - <T> int **binarySearch**(List<? extends T> list, T key, Comparator<? super T> c)
  - Lembre-se que a lista precisa estar ordenada para aplicar busca binária
  - Retorna a posição do elemento na lista ou o negativo de onde o elemento deveria estar
  - Collections.binarySearch(lista, 2);

unesp

Algoritmos sobre Collections

- Composição
  - int frequency(Collection<?> c, Object o)
    - Retorna a quantidade de elementos iguais ao que foi passado
  - boolean disjoint(Collection<?> c1, Collection<?> c2)
    - Retorna **true** se as coleções não tem elementos em comum
- Máximo e mínimo:
  - Métodos que determinam o maior e menor elemento em uma coleção
  - Possuem duas versões: uma considera a ordem natural (**Comparable**) e outra uma ordem específica (**Comparator**)

unesp

Algoritmos sobre Collections

- Classe Arrays
- static <T> List<T> **asList**(T... a)
  - Retorna uma **List** (view) do array passado
  - Não é possível adicionar ou remover elementos
  - Mudanças no array (lista) afetam a lista (array).

```
String[] array = {"João", "Maria", "José"};
List<String> lista = Arrays.asList(array);
System.out.println(lista); // [João, Maria, José]

array[0] = "Pedro";
System.out.println(lista); // [Pedro, Maria, José]

lista.set(1, "Ana");
System.out.println(Arrays.toString(array)); // [Pedro, Ana, José]
```

unesp

Boa prática

- É recomendável sempre declarar tipos de coleções utilizando a interface que o define. Isso proporciona:
  - Flexibilidade para alterar o tipo (apenas a instanciação precisa ser alterada).
  - Garante que apenas operações padrão serão usadas.
- Exemplo

```
Set<String> s = new HashSet<String>();
...
```

  - Se quisermos que o programa tenha os elementos do conjunto 's' de forma ordenada, basta mudar esta linha

```
Set<String> s = new TreeSet<String>();
...
```

unesp

Iteradores

Iteradores

- Objetos do tipo **Iterator** permitem percorrer e remover elementos de uma coleção
- Toda coleção possui um método que retorna um **Iterator**
- Interface **Iterator** tem três métodos

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

Iteradores

- boolean hasNext()
  - Retorna true se há elementos a serem lidos no iterador
- E next()
  - Retorna o próximo elemento do iterador.
- void remove()
  - Remove o último elemento obtido pela chamada de **next()**
  - Só é possível chamar uma vez para cada chamada de **next()**
  - Se essa regra for desrespeitada, uma exceção é lançada.

unesp

Iteradores

- O método **remove()** de **Iterator** é a única maneira segura de alterar uma coleção durante uma iteração

```
Antes: 5 9 12 18 25 55 67 81 83
```

```
static void filterSafe(List<Integer> list) {
    for (Iterator<Integer> it = list.iterator(); it.hasNext(); )
        if (it.next() > 10)
            it.remove();
}
```

```
Depois: 5 9
```

```
static void filterUnsafe(List<Integer> list) {
    for (int i = 0; i < list.size(); i++)
        if (list.get(i) > 10)
            list.remove(i);
}
```

```
Depois: 5 9 18 55 81
```

Iteradores

- Outra vantagem de iteradores é que não dependem do tipo de coleção
  - Nem toda coleção possui um método de remoção por índice como **List**
  - Cada coleção tem sua maneira de percorrer os elementos.
  - A interface **Collection** provê o método **iterator()**
  - Isso permite criar uma solução genérica

```
static void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext(); )
        if (!cond(it.next()))
            it.remove();
}
```

unesp

Criando Novas Coleções

- Muitos programadores nunca precisarão criar suas próprias classes de coleções
  - As classes concretas oferecidas na API Java resolvem muitos problemas.
- Porém, é possível que algum dia você queira implementar sua própria coleção
- Neste caso, é interessante utilizar as classes abstratas para cada tipo de coleção
  - Alguns métodos já estão implementados
  - Outros precisam ser implementados
  - Mesmo os que já estão implementados podem ser sobrescritos

unesp

Criando Novas Coleções

- Classes abstratas de coleções
  - AbstractCollection
  - AbstractSet
  - AbstractList
  - AbstractSequentialList
  - AbstractQueue
  - AbstractMap

unesp

Resumo

- *Generics*
- Classes e métodos genéricos
- Genéricos restritos
- Herança de classes genéricas
- *Wildcards*
- Collections
- Interfaces e classes de coleções
- Algoritmos sobre genéricos
- Iteradores

unesp

Exercício 1

- Crie um aplicativo em Java que faça a leitura de um texto na linha de comando (ou arquivo) e conte a ocorrência de cada palavra no texto
  - Case-insensitive
- Para isso, utilize **Map<K,V>**ou **SortedMap<K,V>**
- Imprima a relação de palavras do texto e suas ocorrências de duas formas
  - Em ordem alfabética
  - Top 10 das palavras que mais ocorrem no texto
- Veja um exemplo de entrada/saída no último slide

unesp

Exercício 1

- Utilizando o que já foi implementado no exercício anterior, escreva a classe **MapUtils** que contém o método genérico:
  - static <K> void printRandomText(Map<K, Integer>)
- Este método deve ser capaz de criar um texto aleatório baseado no conteúdo do **Map**
- Em outras palavras, o método deve criar um texto que contenha todas as chaves K, aparecendo na quantidade exata de vezes.
  - A cada chamada, o método deve gerar um texto diferente.

unesp

Exercício 2

- Crie uma estrutura de dados **árvore de busca binária** para armazenar tipos genéricos
  - Crie a classe **TreeNode**
  - Crie a classe **Tree**
  - Crie um aplicativo que utilize a classe Tree
- Classe **TreeNode**
  - O tipo armazenado em um nó precisa ser **Comparable**
  - Os atributo de um nós são: dado, referência para o filho esquerdo e referência para o filho direito
  - Métodos que achar necessário, segundo sua implementação

unesp

Exercício 2

- Classe **Tree**
  - Precisa ser genérica, pois é o tipo de dado que os nós irão armazenar
  - Seu único atributo é a referência para o nó raiz
  - Implemente um método para Inserção
  - Implemente os métodos de Impressão
    - preOrder
    - inOrder
    - posOrder

unesp

Exercício 1

```
public class Tree<T extends Comparable<T>> {
   . . .

    public void insert(T data) {
        if (root == null) {
            root = new TreeNode<T>(data);
        } else {
            insert(data, root);
        }
    }

    private void insert(T data, TreeNode<T> node) {
        if (data.compareTo(node.getData()) < 0) {
            if (node.getLeftChild() == null) {
                node.setLeftChild(new TreeNode<T>(data));
            } else {
                insert(data, node.getLeftChild());
            }
        } else {
            if (node.getRightChild() == null) {
                node.setRightChild(new TreeNode<T>(data));
            } else {
                insert(data, node.getRightChild());
            }
        }
    }

   . . .

  }
```

Exercício 2

I used to rule the world
Seas would rise when I gave the word
Now in the morning I sleep alone
Sweep the streets I used to own
I used to roll the dice
Feel the fear in my enemyseyes
Listened as the crowd would sing
Now the old king is dead
Long live the king
One minute I held the key
Next the walls were closed on me
And I discovered that my castles stand
Upon pillars of salt and pillars of sand
I hear Jerusalem bells are ringing
Roman Cavalry choirs is singing
Be my mirror my sword and shield
My missionaries in a foreign field
For some reason I cannot explain
Once you would gone there was never
Never an honest word
That was when I ruled the world

Alfab.

a 1
alone 1
an 1
and 3
are 1
as 1
be 1
bells 1
cannot 1
castles 1
calvary1
choirs 1
closed 1
crowd 1
dead 1
...

TOP 10

the 12
i10
my 5
and 3
in 3
to 3
used 3
would 3
is 2
king 2

Bibliografia

- DEITEL, H. M. & DEITEL, P.J. "Java : como programar", Bookman, 2017.
- Material baseado nos slides:
  - Luiz E. Virgilio da Silva. Notas de Aula de Programação Orientada a Objetos (ICMC/USP).
  - José Fernando Junior. Notas de Aula de Programação Orientada a Objetos (ICMC/USP).