Universidade Federal de Campina Grande
Centro de Engenharia Eletrica e Informática
Departamento de Sistemas e Computação
Graduação em Ciência da Computação

Exercícios de Revisão de Java – Generics, Collections, Comparable e Comparator

Objetivo: Experimentar os diferentes tipos abstratos de dados (TADs) disponíveis para a codificação de coleções de dados no *collections framework* de Java. Ao final do experimento, o aluno deve compreender as diferenças semânticas entre listas, conjuntos, pilhas, filas, filas de prioridade e mapas (ou dicionários). Além disso, o aluno deve perceber que o mesmo tipo abstrato de dados admite diferentes implementações e que isso é o fator determinante para a eficiência.

Tipos Genéricos em Java

Leia o tutorial disponibilizado para responder as perguntas seguintes.

1. Para que serve o conceito de tipos genéricos implementados por Java?
2. Suponha que voce deseja definir uma interface A.java contendo um método m que recebe um objeto que pode ter diversos tipos (mas não pode ser diretamente do tipo Object). Implemente uma solução?
3. Suponha que voce deseja estender a interface acima com uma sub-interface B.java que tem o método m funcionando especificamente para objetos do tipo String. Implemente uma solução?
4. Seria possível criar uma classe C.java implementando a interface A.java e mesmo assim a implementação do método m na classe C poderia funcionar para diversos tipos de parâmetros? Implemente uma solução.
5. O que acontece quando declaramos em uma classe parametrizada um método da seguinte forma: `public void m(Collection<T> obj)`?
6. O que acontece quando declaramos em uma classe parametrizada um método da seguinte forma: `public void m(Collection<?> obj)`?
7. O que acontece quando declaramos em uma classe parametrizada um método da seguinte forma: `public void m(? extends T obj)`?
8. O que acontece quando declaramos em uma classe parametrizada um método da seguinte forma: `public void m(? super T obj)`?
9. Agora é hora de generalizar a coleção implementada no exercício anterior (RepositorioProdutos). Deixe a coleção genérica para que ela aceite outros tipos além de Produto.
10. Em seguida adapte suas classes RepositorioProdutosArray e RepositorioProdutosArrayList para implementarem a coleção genérica funcionando apenas para objetos do tipo Produto.
11. Observe a classe Vetor.java, que representa uma implementação de um vetor. Note que a classe trabalha com o tipo Object, tornando-a capaz de guardar qualquer tipo, sem usar Generics. Entretanto, não é possível fazer atribuições específicas sem uso de cast, como por exemplo: String nome = vetor.procurar("Junior"). Isso acontece porque o método procurar retorna um Object.
    a. Altere a classe Vetor para que ela trabalhe com um tipo genérico.
    b. Em seguida, implemente/modifique o corpo dos métodos para que eles fiquem realmente implementados como devem funcionar (de acordo com os comentários).
12. Agora, reutilizando seu trabalho no exercício de revisão sobre arrays e interfaces, generalize sua implementação de repositório de produtos utilizando os conceitos de generics. Agora seu repositório deverá ser capaz de guardar qualquer tipo de produto, utilizando o recurso de generics.

a. A primeira coisa a fazer é sobrescrever as classes importadas no projeto com as suas implementações
b. Depois concentre-se em usar generics na implementação de repositório de produtos.

Uso de Comparable e Comparator

Leia a documentação das interfaces Comparable e Comparator. Eles podem ser usados para comparar elementos da seguinte forma:

- Comparable – define um objeto que é comparavel com outro objeto. Ao implementar a interface Comparable, uma classe T precisa apenas redefinir o método compareTo(T obj).
- Comparator – estabelece um componente externo a ser usado na comparação de objetos. Por exemplo, uma classe que deseja comaprar objetos do tipo T (que não são Comparable) precisa fazer uso de uma implementação de um Comparator para comparar esses objetos do tipo T. Dessa forma, o desenvolvedor precisa usar ou definir algum sub-tipo de Comparator, com o método compare(T o1, To 2) que compara objetos do tipo T.

1. Imagine agora que você deseja tornar a classe Vetor (genérica) capaz de guardar apenas tipos que sejam comparáveis. Altere a classe Vetor para que isso seja possível. Dica: você precisará apenas de uma pequena modificação no parâmetro da classe (no tipo genérico manipulado pela classe).
2. Uma vez que o Vetor pode guardar elementos comparáveis, é possível encontrar elemento máximo e mínimo dentro do vetor.
    a. Implemente um método que retorna o elemento máximo do vetor **maximo()**.
    b. Implemente um método que retorna o elemento minimo do vetor **minimo()**.
3. Imagine agora que voce deseja que a busca pelos elementos máximo e mínimo sejam feitos por um comparator à parte (um outro objeto).
    a. Implemente dois comparadores (um usado para encontrar o máximo e outro para encontrar o mínimo). Eles deverão ser usados pela classe Vetor. Obs: essa implementação pode ser feita dentro do próprio arquivo Vetor.java, após a classe Vetor. Isso gerará uma classe com visibilidade restrita (mas não inner class!!!).
    b. Na classe TestarVetor, instancie um Vetor que guardará objetos do tipo Aluno.
    c. Realize testes para ver se o funcionamento do comparator está de acordo.

Note que o funcionamento do máximo e mínimo depende do comparator. Como eles possuem uma lógica oposta, se você inverter os comparators, os métodos irão funcionar exatamente ao contrário. Isso é uma padrão de projeto muito útil para a disciplina no futuro!

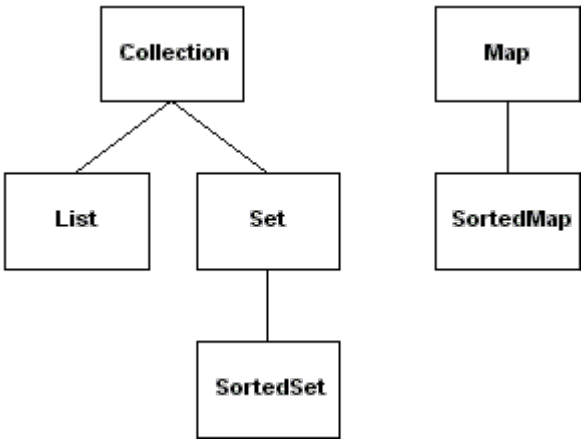
Collections

Sempre que precisamos que um programa armazene um número não pré-determinado de objetos na memória, temos que usar o conceito de *coleção de dados*. Em termos de programação, uma coleção é um objeto que é capaz de armazenar referências de outros objetos. Eis os tipos clássicos de coleções com que normalmente tratamos em um programa:

1. coleções;
2. conjuntos;
3. listas ou seqüências;
4. pilhas;

5. filas;
6. filas de prioridade;
7. mapas;
8. conjuntos e mapas ordenados.

Java oferece um ótimo suporte a coleções. De fato, todas as coleções acima podem ser usadas em programas Java sem nenhum esforço de programação. A figura abaixo apresenta uma forma simplificada da hierarquia de interfaces e classes do *framework* de coleções de Java.



How to choose which Java collection class to use?

The Java Collections API provides a whole host of data structures, especially since the API was expanded in Java 5 (and again slightly in Java 6) to include concurrent collections. At first, the array of choices can be a little daunting: should I use a `HashMap` or a `LinkedHashMap`? When should I use a list or a `HashSet`? When should I use a `TreeMap` rather than a `HashMap`? But with a bit of guidance, the choice needn't be quite so daunting. There are also a few cases where it's difficult to decide because the choice is very arguable. And in other cases, having a clear set of rules of thumb can guide you to an appropriate decision.

Basic approach to choosing a collection

The overall approach I'd suggest for choosing is as follows:

1. choose the **general type** of organisation that your data needs to have (e.g. map or list); without too much thought, this is usually fairly clear;
2. then, choose the implementation of that type that has the **minimum functionality** that you actually require (e.g. don't choose a sorted structure if you don't actually need the data to be sorted).

In general, the algorithm that underlies each collection class is designed to be a good tradeoff between efficiency and certain minimal requirements. So as long as you understand the *minimal requirements* that a given class is designed to provide, you shouldn't need to get too bogged down in the actual algorithms (though if you *are* interested in algorithms, the source code to all the collections classes is available and they make fascinating case studies, of course).

1. Basic collection types

The first part of the decision is choosing what *"basic category"* of organisation or functionality your data needs to have. The broad types are as follows:

Collection type	Functionality	Typical uses
List	• Essentially a variable-size array; • You can usually add/remove items at any arbitrary position; • The order of the items is well defined (i.e. you can say what position a given item goes in in the list).	Most cases where you just need to store or iterate through a "bunch of things" and later iterate through them.
Set	• Things can be "there or not"—when you add items to a set, there's no notion of *how many times* the item was added, and usually no notion of ordering.	• Remembering "which items you've already processed", e.g. when doing a web crawl; • Making other *yes-no decisions* about an item, e.g. "is the item a word of English", "is the item in the database?" , "is the item in this category?" etc.
Map	• Stores an *association* or mapping between "keys" and "values"	Used in cases where you need to say "for a given X, what is the Y"? It is often useful for implementing in-memory caches or indexes. For example: • For a given user ID, what is their cached name/User object? • For a given IP address, what is the cached country code? • For a given string, how many instances have I seen?
Queue	• Like a list, but where you **only ever access the *ends*** of the list (typically, you add to one end and remove from the other).	• Often used in **managing tasks** performed by different threads in an application (e.g. one thread receives incomming connections and puts them on a queue; other "worker" threads take connections off the queue for processing); • For **traversing hierarchical structures** such as a filing system, or in general where you need to remember **"what data to process next"**, whilst also adding to that list of data; • Related to the previous point, queues crop up in various algorithms, e.g. build the encoding tree for *Huffman compression*.

The first three of these are really "bread and butter" collection types. On the other hand, it's probably fair to say that queues will be used less often by many programmers. Even if you are writing a multithreaded server or application involving multi-threaded job processing, you can often use the Java 5 Executor framework (which uses queues underlyingly) without needing to manipulate queues directly.

1. Leia as especificações de Collection, Set e List. Explique brevemente a semântica de cada uma.
2. A classe ArrayList, que você já deve conhecer, implementa a interface List<E>. Essa relação permite que declaremos a variável lista de duas formas diferentes: List<E> lista = new ArrayList<E>() ou ArrayList<E> lista = new ArrayList<E>(). Qual das duas formas é preferível no caso geral? E por quê?.