MC322
Primeiro semestre de 2022

Laboratório 5

---

1 Objetivo

O objetivo deste laboratório será a familiarização com as outras coleções disponíveis no Java. Você pode usar a classe **RandomStringGen.java** se achar util. Ela gera strings aleatórias de um tamanho determinado.

2 O operador == e o método Object#equals

2.1 Introdução

A habilidade de comparar duas variáveis é essencial na programação de computadores. Na linguagem C, o operador *equals* (i.e. ==) é frequentemente empregado a fim disto:

```c
void say_if_equals(int a, int b) {
    if (a == b) {
        printf("a is equal to b!");
    } else {
        printf("Not so lucky");
    }
}
```

Listing 1: A comparação de dois inteiros na linguagem C.

A situação fica um pouco mais complexa ao comparar arrays, já que o valor contido em uma variável do tipo `int*` é uma posição de memória e comparar duas posições de memória distintas sempre retorna falso – ou 0 –, mesmo que os arrays contenha elementos idênticos.

2.1.1 O método *equals* em Java

No Java, um "problema" similar ocorre em relação aos objetos. Assim como os arrays, os valores contidos em variáveis – ou atributos – associadas à classes contém apenas referências à objetos daquelas classes.

O operador '==' de Java se limita a comparar se duas variáveis referenciadas apontam para um mesmo objeto em memória.

A utilização do método *equals* conforme implementado em Object, utiliza o operador '==' e portanto se limita à comparação das referências e não dos dados propriamente contidos nos objetos.

O código abaixo ilustra exemplos deste "problema":

```
> Integer.valueOf(10) == Integer.valueOf(10)
true
> "Hello world!" == "Hello world!"
true
> new Integer(10) == new Integer(10)
false
> new String("Hello world!") == new String("Hello world!")
false
> Arrays.asList("Mariah", "James", "Smith")
    == Arrays.asList("Mariah", "James", "Smith")
false
```

Listing 2: Comparações entre objetos utilizando o operador *equals*.

Nota: é importante mencionar que as sentenças `Integer.valueOf(10) == Integer.valueOf(10)` e `"Hello world!" == "Hello world!"` resultam em `true` pois existem mecanismos de cache ativos capazes de identificar que objetos da classes **String** e **Integer** contendo os valores "Hello World!"e 10, respectivamente, foram previamente construídos. Tais mecanismos, portanto, simplesmente retornam referências à estes objetos. Nos outros exemplos são efetivamente criados objetos distintos em memória e por isso a comparação devolve `false`.

2.1.2 Object#equals(Object obj)

A fim de se comparar objetos de forma mais completa, utilizamos então o método `Object#equals(Object obj)`. Como definido em **Object**, dois objetos são iguais se as referências que os indicam são iguais. Isto é, `a.equals(b)` é simplesmente um *alias* para `(a == b)`. Este método pode, entretanto, ser sobrescrito de forma similar ao método `Object#toString()`. No exemplo abaixo, duas pessoas são iguais se elas apresentam o mesmo identificador e nome:

```
public class Person {
  private int id;
  private String nome;

  public Person(int id, String name) {
    this.id = id;
    this.name = name;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj) return true;
    if (!(obj instanceof Person)) return false;
    Person p = (Person) obj;
    if (name == null || p.name == null) return false;
    return id == p.id && name.equals(p.name);
  }
}
```

Listing 3: Sobrescrita de *Object#equals(Object obj)* na classe **Person**.

Note que muitas classes já sobrescrevem `Object#equals(Object obj)`, como **Integer**, **String** ou **AbstractList**:

```
> new Integer.valueOf(10).equals(Integer.valueOf(10))
true
> new Integer(10).equals(new Integer(10))
true
> "Hello world!".equals("Hello world!")
true
> new String("Hello world!").equals(new String("Hello world!"))
true
> Arrays.asList("Mariah", "James", "Smith").equals(
    Arrays.asList("Mariah", "James", "Smith"))
true
```

Listing 4: Comparações entre objetos utilizando o método *equals*.

Algumas classes já utilizam o `Object#equals(Object obj)` internamente:

```
> List<String> names = Arrays.asList("Mariah", "James", "Smith")
> names.contains(new String("James"))
true
> names.indexOf(new String("James"))
1
> names.contains(new String("Kevin"))
false
> names.indexOf(new String("Kevin"))
-1
```

Listing 5: Exemplos de métodos em **List<String>** que utilizam String#equals(Object obj) internamente.

2.2 Atividade

1. Inicie um novo projeto Java chamado Lab5RAxxxxxx, onde xxxxxx deve ser substituído pelo seu RA.
2. Reutilize as classes definidas no pacote *base* do laboratório 4 (e.g. **Carta**, **Baralho**) no projeto Lab5.
3. Sobrescreva o método Object#equals(Object obj) na classe **Carta**, considerando seus atributos. Dica: não é suficiente simplesmente comparar o nome das cartas, visto que um único baralho pode conter mais de uma carta com o mesmo nome.
4. Sobrescreva o método Object#equals(Object obj) na classe **Baralho**, considerando as cartas presentes e sua ordenação.

2.3 Tarefas

1. Crie uma classe Main.java onde primeiramente é criado um **ArrayList** de **Cartas** (ArrayList<Carta> cartas). Adicione 10.000 objetos aleatórios de subclasses de **Carta** à esta lista. Qual é a soma do tempo necessário para buscar cada elemento em cada posição i da lista utilizando List#get(int i)? Faça o programa imprimir este tempo.
2. Na mesma classe main crie uma **LinkedList** (LinkedList<Carta> cartas) e imprima o tempo necessário para acessar cada elemento como no item anterior.
3. Imprima qual é o tempo necessário para buscar todos os elementos o contidos na lista cartas (ArrayList<Carta> cartas) através do método List#contains(Object o)?
4. Faça a mesma coisa que o item anterior mas utilizando a **LinkedList** (LinkedList<Carta> cartas) e imprima o tempo necessário.
5. Uma lista (**ArrayList**, **LinkedList**) aceita uma carta repetida?

Dica: o tempo transcorrido pode ser calculado como descrito em Lst. 6.

```
long s = System.nanoTime();

// Processamento...

System.out.println("A operacao demorou " + (System.nanoTime() - s) / 1000000 + " ms"));
```

Listing 6: Medindo tempo transcorrido em um trecho de código.

3 Outras coleções, comparadores e Object#hashCode()

3.1 Introdução

Listas são coleções necessárias quando a ordem dos elementos ali contidos é um fator determinante na solução do problema em mãos. A ordenação dos elementos de uma lista segue a ordem de inserção, sendo independente de qualquer propriedade do objeto ali contido. `List#contains(Object o)` e `List#indexOf(Object o)` devem, portanto, navegar por cada elemento *el* contido na lista e verificar o resultado da operação `el.equals(o)`; o que pode resultar em baixa performance em listas contendo imensa quantidade de objetos.

Quando manipulando grandes quantidades de objetos – e a ordem de inserção pouco importa –, outras coleções (e.g. **TreeSet**, **HashSet**) podem ser mais adequadas.

3.1.1 HashSet

O **HashSet** é uma implementação de **Set** (que por sua vez implementa **Collection**) que utiliza internamente uma tabela hash para guardar os objetos ali inseridos. Esta implementação é interessante quando a quantidade e frequência de acesso à elementos inseridos são altas e ordenação destes não importa.

Como usualmente, uma tabela hash se utiliza de uma função hash $h\colon T \to \mathbb{N}$ que mapeia objetos de uma classe T à uma posição na tabela. Por padrão, a implementação da função hash utilizada é o método `Object#hashCode()`; que pode ser sobrescrita da mesma forma que o método `Object#equals(Object obj)`.

O exemplo abaixo descreve uma possível implementação para a função hash da classe **Person**, considerando sua *identidade* e *nome* (os atributos considerados em `Person#equals(Object obj)`).

```
1  import java.util.Objects;
2
3  public class Person {
4
5    // ...
6
7    @Override
8    public int hashCode() {
9      Objects.hash(name, age);
10   }
11 }
```

Listing 7: Uma possível implementação de função hashCode para a classe **Person**.

3.1.2 TreeSet

O **TreeSet** é uma implementação de **Set** (que por sua vez implementa **Collection**) que utiliza internamente uma árvore rubro negra (i.e. um tipo de árvore binária auto balanceada). Essa estrutura apresenta uma performance em acesso inferior ao HashSet, mas garante que os elementos estejam sempre ordenados de acordo com um determinado comparador.

Dado a natureza das árvores binárias, onde objetos são colocados em nós à esquerda (menor) ou direita (maior) de um objeto contido em um nó já existente, precisamos primeiramente descrever o que significa para um objeto de uma classe ser maior, menor ou igual à um outro desta mesma classe. Isto pode ser feito fazendo com que a classe implemente a interface **Comparable**.

Por exemplo, podemos criar um **TreeSet** de **Persons** indexados em ordem crescente, primeiro pelo nome e segundo por idade (se tiverem nomes iguais o desempate é por idade).

```
1  import java.util.Objects;
2
3  public class Person implements Comparable<Person>{
4      String name;
```

```
5       int age;
6
7       ...
8       @Override
9       public int compareTo(Person o){
10          if (this.equals(o)) return 0;
11
12          if (this.name.equals(o.name)){
13              if (this.age < o.age)
14                  return -1;
15              else
16                  return 1;
17          }
18          return this.name.compareTo(o.name);
19      }
20
21  }
```

Listing 8: Um comparador de identidades de pessoas.

Finalmente, percorrer o conjunto *persons* com o for reduzido (`for (Person p : persons) {}`) resultaria em um fluxo de objetos da classe **Person** ordenados.

3.2 Atividade

1. Sobrescreva o método Object#hashCode() na classe **Carta**, considerando seus atributos. Dica: algumas classes já sobrescrevem seus próprios hashes (e.g. UUID#hashCode(), String#hashCode()).

2. Sobrescreva o método Object#hashCode() na classe **Baralho**.

3.3 Tarefas

1. Crie uma classe Main2.java.

2. Crie um **HashSet** de **Cartas** adicione 10.000 objetos aleatórios de subclasses de **Carta** à esta coleção. Imprima qual é o tempo necessário para buscar todos os elementos *o* contidos na lista cartas através do método List#contains(Object o)?

3. Crie uma **TreeSet** de **Cartas** e como no item anterior adicione 10000 objetos aleatórios e imprima o tempo necessário para buscar os objetos. Nota: o comparador empregado aqui deve utilizar os mesmos atributos considerados no método Carta#equals(Object obj).

4. Um conjunto (**HashSet**, **TreeSet**) aceita uma carta repetida?

4 Submissão

Submeta no google classroom a pasta do projeto comprimida como um arquivo zip, de tal forma que o arquivo se chame Lab5RAxxxxxx.zip (xxxxxx deve ser substituído pelo seu RA). A pasta deve ter sido preparada conforme especificado.