MC302
Primeiro semestre de 2017

Laboratório 6

Professores: Esther Colombini (esther@ic.unicamp.br) e Fábio Luiz Usberti (fusberti@ic.unicamp.br)
PEDs: (Turmas ABCD) Elisangela Santos (ra149781@students.ic.unicamp.br), Lucas Faloni (lucasfaloni@gmail.com), Lucas David (lucasolivdavid@gmail.com), Wellington Moura (wellington.tylon@hotmail.com)
PEDs (Turmas EF) Natanael Ramos (naelr8@gmail.com), Rafael Arakaki (rafaelkendyarakaki@gmail.com)
PAD: (Turmas ABCD) Igor Torrente (igortorrente@hotmail.com)
PAD: (Turmas EF) Bleno Claus (blenoclaus@gmail.com)

---

1 Objetivo

O objetivo deste laboratório será a familiarização com as outras coleções disponíveis no Java e a API funcional disponível para streams implementada a partir do Java 8.

2 O operador *equals* e 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 operador *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. A utilização do *equals* se limitaria, portanto, 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
> new Integer(10) == new Integer(10)
false
```

```
> "Hello world!" == "Hello world!"
true
> 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.

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 Lab6.
2. Reutilize as classes definidas no pacote *base* do laboratório 4 (e.g. **Carta**, **Baralho**) no projeto Lab6.
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. Tente fazê-lo em uma única linha.

2.3 Tarefas

1. Crie um **ArrayList** de **Cartas** (ArrayList<Carta> cartas) e adicione 10000 cartas à esta lista. Qual é a soma do tempo necessário para buscar o elemento em cada posição *i* da lista utilizando List#get(int i)?
2. Repita o experimento anterior utilizando uma **LinkedList** (LinkedList<Carta> cartas) e reporte o tempo necessário.
3. Novamente, crie um **ArrayList** (ArrayList<Carta> cartas) de **Cartas** e adicione 10000 cartas à esta lista. Qual é o tempo necessário para buscar todos os elementos *o* contidos na lista cartas através do método List#contains(Object o)?
4. Repita o experimento anterior utilizando uma **LinkedList** (LinkedList<Carta> cartas) e reporte o tempo necessário.
5. Uma lista (**ArrayList**, **LinkedList** ou **Vector**) 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  public class Person {
2
3    // ...
4
5    @Override
6    public int hashCode() {
7        int hash = 3;
8        hash = 67 * hash + id;
9        hash = 13 * hash + (name != null ? name.hashCode() : 0);
10       return hash;
11   }
12 }
```

Listing 7: Uma possível implementação de função hash 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 com a implementação de uma *lambda function* f que possui dois argumentos (os objetos a e b que devem ser comparados) e retorna um valor negativo se a for menor que b, 0 se a for igual à b e um valor positivo se a for maior que b.

Por exemplo, podemos criar um **TreeSet** de **Persons** indexados em ordem crescente por seus identificadores. As sentenças seguintes fazem exatamente isso e são equivalentes:

```java
Collection<Person> persons = new TreeSet<>((Person p1, Person p2) -> {
  return p1.getId() - p2.getId();
});
// Ou...
Collection<Person> persons = new TreeSet<>((p1, p2) -> {
  return p1.getId() - p2.getId();
});
// Ou...
Collection<Person> persons = new TreeSet<>((p1, p2) -> p1.getId() - p2.getId());
// Ou...
Collection<Person> persons = new TreeSet<>(Comparator.comparingInt(Person::getId));
```

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 por sua identidade.

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**. Tente fazê-lo em uma única linha.

3.3 Tarefas

1. Crie um **HashSet** de **Cartas** e adicione 10000 cartas à esta coleção. Qual é o tempo necessário para buscar todos os elementos *o* contidos na lista cartas através do método `List#contains(Object o)`?

2. Repita o experimento anterior utilizando uma **TreeSet** de **Cartas** e reporte o tempo necessário. Nota: o comparador empregado aqui deve utilizar os mesmos atributos considerados no método `Carta#equals(Object obj)`.

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

4 A *API* Stream

Todas as coleções implementam, a partir do Java 8, o método `Collection#stream()`, que retorna um **Stream** (ou fluxo) de objetos contidos naquela coleção. O fluxo depende fundamentalmente de como a coleção se organiza internamente, mas sempre pode ser utilizado para processar os dados ali contidos de forma mais sucinta.

Seja *people* uma coleção de pessoas como definida abaixo:

```java
Collection<Person> people = Arrays.asList(
    new Person(0, "John Reese"), new Person(1, "Kelly Sanches"),
    new Person(2, "Jennifer Walker"), new Person(3, "John Creedence"));
```

Listing 9: Lista *people* de objetos da classe **Person**.

Alguns exemplos de operações possíveis sobre streamings de *people*:

```
Collection<Person> johns = people.stream()
  // Filtre o fluxo, capturando somente quem se chama John.
  .filter(p -> p.getName().startsWith("John"))
  // Ordene decrescentemente por seu id.
  .sort((p1, p2) -> p2.getId() - p1.getId())
  // Ignore os dois primeiros johns.
  .skip(2)
  // Limite o numero de elementos no fluxo por 5.
  .limit(5)
  // Construa uma lista a partir do fluxo.
  .collect(Collectors.toList());
```

Listing 10: Exemplo de filtragem, ordenação e limitação.

```
final int maxId = 2;
Collection<Person> somePeople = people.stream()
  .filter(p -> p.getId() < maxId)
  .collect(Collectors.toList());
```

Listing 11: Exemplo de filtragem com variáveis. Note que uma variável precisa ser final ou efetivamente final para ser usada dentro da função *lambda*.

```
Person kelly = people.stream()
  // Filtre o fluxo, capturando somente quem se chama Kelly.
  .filter(p -> p.getName().startsWith("Kelly"))
  // Pegue o primeiro objeto.
  .findFirst()
  // Ou retorne null se nenhum existir.
  .orElse(null);
```

Listing 12: Exemplo de seleção.

```
Collection<String> peoplesNames = people.stream()
  // Mapeie cada objeto Person para um objeto String.
  .map(p -> p.getName())
  // Construa uma lista a partir do fluxo.
  .collect(Collectors.toList());
```

Listing 13: Exemplo de mapeamento.

```
integer sumOfPeoplesIds = people.stream()
  // Mapeie cada objeto Person para um objeto Integer
  // (transformando Stream<Person> em um IntStream).
  .mapToInt(p -> p.getId())
  // Some todos as identidades.
  .sum();
```

Listing 14: Exemplo de redução.

4.1 Atividades

1. Crie uma coleção com multiplos objetos das sub-classes de **Carta**.
2. Utilize o *stream* da coleção criada para selecionar o Lacaio de maior ataque.
3. Utilize o *stream* da coleção para calcular a soma de pontos de ataque dos lacaios presentes.
4. Utilize o *stream* da coleção para ordenar os lacaios por seus pontos de vida.

5 Submissão

Para submeter a atividade, utilize o Moodle (`https://www.ggte.unicamp.br/ea`). Crie um arquivo texto com as respostas para cada item da seção tarefas e as saídas geradas pelo código. Compacte o código-fonte contido no diretório **src** juntamente com arquivo de respostas no formato .zip ou similar e nomeie-o **Lab6-000000.zip**, trocando '000000' pelo seu número de RA. Submeta o arquivo na seção correspondente para esse laboratório no moodle da disciplina MC302.

Datas de entrega

- Dia **02/05** Turma **ABCD** até às 23:55h
- Dia **06/05** Turma **EF** até às 23:55h