Professor J

Igualdade em Java

Leandro Yung
(leandro@expressojava.com.br) é formado em engenharia elétrica pela USP. Analista-programador na Claro, instrutor de Java da Sun Microsystems e professor de pós-graduação da FIAP e IBTA, na área de Java, XML, WebServices e SOA.

Pequenos cuidados tomados logo no início, durante a criação de classes, podem evitar dores de cabeça com a execução e funcionamento de aplicações inteiras feitas em Java. A simples comparação de objetos criados a partir de uma mesma classe pode ser um grande problema, seja por não comparar corretamente os objetos seja por funcionar inadequadamente, lançando até exceções nas aplicações nos casos mais graves. Iremos abordar no artigo desta edição o uso correto do método equals(). Apresentaremos as principais dicas e regras para garantir um bom funcionamento de toda a aplicação e assim poderemos dizer se um objeto em questão é ou não é idêntico ao de referência.

Introdução

Para o iniciante em Java, o processo de verificar se um objeto é igual a outro é direto para tipos primitivos em Java (uso do operador ==), simples para objetos da classe String (uso do método equals()) e obscuro para classes. Junto ao uso do método equals(), temos o uso do método hashCode().

Na palestra de Danival, do CPqD, durante um evento do JavaAmericana, ele deu algumas orientações e indicou o excelente livro Effective Java do Joshua Blockfoi. Foi nesse livro que eu me inspirei para escrever este artigo.

Dados e Objetos

O processo de comparar dois objetos, aparentemente simples, esconde uma complexidade em função das estruturas dos objetos que serão comparados. Assim, não estaremos comparando estruturas simples, mas estruturas que poderão conter outras. Ou seja, analisaremos objetos que contenham outros.

A simples verificação de igualdade é uma fonte potencial de problemas, podendo gerar funcionamento não previsto do programa. Ao comparar dois objetos, conseguiremos obter resultado de desigualdade para objetos idênticos quanto ao conteúdo, ou pior, podemos gerar uma exceção ou até erro. A aplicação pode, assim, até parar a sua execução.

Para compreender a maneira que esse processo é realizado, precisamos entender a maneira como os dados e objetos são mantidos na memória para uso na linguagem Java. Dessa maneira, necessitaremos antes de uma introdução aos termos memória Stack (pilha) e Heap (de objetos).

Memória Stack e Heap

O Java, como a maioria das linguagens, mapeia as estruturas de dados usadas durante a execução do programa na memória em duas partes: a chamada memória Stack e a memória Heap.

A memória Stack (pilha) recebe esse nome por ter uma estrutura organizada de maneira seqüencial, com dados colocados um sobre o outro. Ela guarda os valores de dados de tipo primitivo e as referências (maneira de se encontrar objetos), aos tipos de dados mais complexos.

A memória Heap, que significa amontoado, acumulado, é mais ampla, não ordenada, que pode ser acessada de maneira direta, para tanto, é necessário saber onde está o dado desejado, o que é providenciado pela referência guardada, na memória Stack.

Assim, a memória Heap contém os objetos na linguagem Java, enquanto a Stack contém os dados primitivos e as referências para os objetos armazenados na Heap.

Vamos fazer um programa simples para ilustrar as memórias Stack e Heap:

Listagem 1. AClass.java

```
public class AClass {
   private int a;

   public void setA(int a) {
      this.a = a;
   }

   public int getA() {
      return a;
   }
}
```

Na listagem 1, temos uma classe simples, que possui um atributo de nome "a" e possui dois métodos (setA() e getA()).

Na listagem 2, temos uma aplicação executável que possui quatro variáveis: um, valor, dois e três. As variáveis um, dois e três são da classe AClass e a variável valor é do tipo primitivo int.

Listagem 2. StackHeap.java

```
public class StackHeap {
   public static void main(String[] args) {
      AClass um   = new AClass();
      int valor   = 15;
      AClass dois = new AClass();
      AClass tres = um;

      System.out.println("Objeto um:" + um);
      System.out.println("Objeto dois:" + dois);
      System.out.println("Objeto tres:" + tres);
      System.out.println("Valor:" + valor);
   }
}
```

O programa irá instanciar (criar) dois objetos da classe AClass e atribuir (guardar a referência) nas variáveis um e dois. Depois a variável "tres" irá receber o conteúdo da variável "um", ou seja, a referência ao objeto contido em "um".

Após esse processo, o programa irá apresentar na tela, os valores guardados por cada variável. Obtemos assim uma saída parecida com a figura 1.

```
Objeto um:AClass@119c082
Objeto dois:AClass@1add2dd
Objeto tres:AClass@119c082
Valor:15
```

Figura 1. Saída da execução do programa StackHeap.

Para entendermos melhor, apresentamos na figura 2 um gráfico representativo da execução do programa acima.

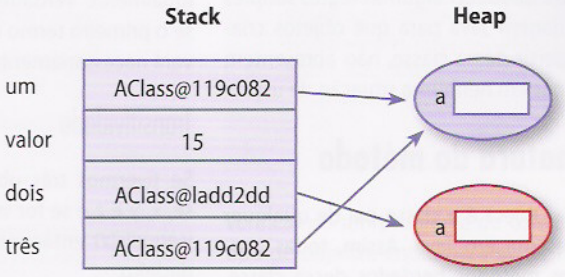

Figura 2. Representação Gráfica.

Podemos ver que o Java irá guardar no Stack os valores de variáveis do tipo primitivo (número, caractere, booleano). Porém, para variáveis que referenciem objetos, teremos um número de referência, que indicará onde encontrar o objeto na memória Heap.

Outro ponto interessante de se notar, é que as variáveis "um" e "três" contêm a mesma referência e apontam para o mesmo objeto, assim, caso esse objeto seja alterado, terá impacto em ambas.

Igualdade – Uma breve descrição

Um objeto pode ser referenciado por diversas variáveis, e assim, observamos que todos os valores armazenados na memória Stack terão a mesma referência, "endereço" do objeto. Assim, os valores armazenados na Stack, ou seja, o conteúdo direto de cada da variável, quando comparados, serão a mesma referência, o que resulta em igualdade.

No entanto, podemos ter dois objetos distintos que possuam o mesmo valor para os dados internos, assim, a variável que contém a referência ao primeiro terá valor diferente da variável que contém o segundo. Esse resultado é indesejável, já que os objetos deveriam ser considerados iguais.

Para tanto, as classes Java possuem o método equals() que avalia internamente os objetos e retorna verdadeiro; caso os objetos comparados possuam os mesmos dados relevantes e falso e caso os objetos comparados tenham alguma diferença.

A maneira que isso é feito deve ser dita pelo programador da classe. Assim, para cada classe devemos criar um método equals(). A linguagem Java provê o método equals() para as classes disponíveis na API (ex.: String, ArrayList, etc.). No entanto, para as classes montadas pelo programador, esse deve prover o seu próprio método para verificar, atributo a atributo, os valores internos relevantes.

Então, ao criar o método equals(), dizemos ao Java como deverá ser feito a verificação da igualdade entre dois objetos distintos. Comparando os atributos de interesse e retornando verdadeiro; caso todos sejam idênticos.

Não basta apenas comparar dois objetos utilizando o operador (==), devemos tomar cuidado e obedecer algumas regras simples da linguagem Java para que objetos criados, a partir dessa classe, não apresentem problemas em nenhuma situação de uso.

Assinatura do método

A classe Object é a classe-pai de qualquer classe escrita em Java. Assim, todas possuem os métodos herdados dessa classe. Nela, está definida a forma que devemos usar o método verificador de igualdade, equals.

Para obtermos sucesso em uma comparação, criamos o método na nossa classe que irá prover a funcionalidade de comparar se um objeto recebido é igual ao atual. A assinatura do método é:

```
public boolean equals(Object obj)
```

Assim, temos um método visível publicamente, que recebe um objeto de qualquer classe e retorna verdadeiro; caso o objeto comparado seja igual ao objeto atual e falso; caso contrário.

Regras a serem obedecidas

Existem regras de determinação de igualdade definidas na própria especificação da API do Java, no método equals() definido para a classe Object.

Assim, para qualquer objeto não-nulo em Java, obedecemos as seguintes regras:

1. reflexividade;
2. simetria;
3. transitividade;
4. consistência;
5. desigualdade ao nulo.

Reflexividade

Sempre um objeto deve ser igual a ele mesmo. Assim, a regra de comparação deve ser consistente, ou seja, sempre para um dado objeto x devemos ter x.equals(x) como sendo verdadeiro.

Simetria

Tendo dois objetos de mesma classe x e y e se por um lado x.equals(y) for verdadeiro obrigatoriamente y.equals(x) deve ser igualmente verdadeiro. Da mesma forma, se o primeiro termo for falso o segundo deverá necessariamente ser falso.

Transitividade

Se tivermos três objetos da mesma classe, x, y e z e se for verdadeiro x.equals(y) e y.equals(z) então x.equals(z) deve ser verdadeiro.

Consistência

Sempre que fizermos uma comparação entre objetos, caso eles não mudem, o resultado da comparação também não deve mudar.

Desigualdade ao nulo

Qualquer objeto não-nulo, quando comparado com nulo, deve resultar em falso.

Outros fatores

Apesar das regras serem simples, outros fatores devem ser levados em conta, como por exemplo: o método equals() não pode lançar exceção, deve apenas retornar verdadeiro ou falso para uma dada comparação. Esse fato, já complica bastante a vida do programador iniciante.

Práticas

Nada melhor para o entendimento do que a prática e execução. Logo, vamos montar nossas classes e verificar como se comportam diante das implementações.

Montamos a classe Pessoa (conforme listagem 3), que possui dois atributos: nome e sobrenome. Bem como os métodos de acesso: getNome(), setNome(), getSobrenome() e setSobrenome().

Listagem 3. Pessoa.java

```
public class Pessoa {
    private String nome;
    private String sobrenome;

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getSobrenome() {
        return sobrenome;
    }

    public void setSobrenome(String sobrenome) {
        this.sobrenome = sobrenome;
    }
}
```

Vamos montar uma aplicação que usa objetos dessa classe e compara os mesmos. Conforme listagem 4.

Listagem 4. UtilizaPessoa.java

```
public class UtilizaPessoa {
    public static void main (String [] args) {
        Pessoa fulano = new Pessoa();
        Pessoa sicrano = new Pessoa();

        fulano.setNome("João");
        fulano.setSobrenome("Silva");

        sicrano.setNome("Maria");
        sicrano.setSobrenome("Joaquina");

        System.out.println(fulano);
        System.out.println(sicrano);
        System.out.println(fulano.equals(sicrano));
    }
}
```

Executando a classe UtilizaPessoa.java temos um resultado semelhante ao da figura 3.

```
Pessoa@119c082
Pessoa@1add2dd
false
```

Figura 3. Execução de UtilizaPessoa

A comparação resultou em falso, dado que fulano é diferente que sicrano, ou seja, funcionou. No entanto, precisaremos ver como se comporta para objetos de mesmo conteúdo, antes de podermos comemorar o resultado. Assim, altere a classe UtilizaPessoa conforme a listagem 5.

Listagem 5. UtilizaPessoa.java

```
public class UtilizaPessoa {
   public static void main (String [] args) {
      Pessoa fulano = new Pessoa();
      Pessoa sicrano = new Pessoa();

      fulano.setNome("João");
      fulano.setSobrenome("Silva");

      sicrano.setNome("João");
      sicrano.setSobrenome("Silva");

      System.out.println(fulano);
      System.out.println(sicrano);
      System.out.println(fulano.equals(sicrano));
   }
}
```

Agora, ambos objetos, apesar de criados em tempos distintos, apresentam os mesmos dados (nome e sobrenome), assim, esperamos que sejam considerados iguais. Executando novamente, teremos algo semelhante à figura 4.

```
Pessoa@119c082
Pessoa@1add2dd
false
```

Figura 4. Execução de UtilizaPessoa

Vemos então que não funcionou corretamente. Ambos objetos possuem os mesmos dados internos, mesmo nome e sobrenome. No entanto, o método equals() é herdado da classe Object e realiza uma comparação das referências dos objetos e não a comparação de atributo a atributo.

Como o método herdado da classe Object não satisfaz as nossas necessidades, precisamos criar nosso próprio método equals() dentro da classe Pessoa. O leitor pode dizer, nada mais simples, basta comparar cada atributo existente com o atributo do objeto passado para verificar a igualdade, poderíamos ter então uma implementação como a da listagem 6.

Listagem 6. Pessoa.Java

```
public class Pessoa {
   private String nome;
   private String sobrenome;

   public String getNome() {
      return nome;
   }

   public void setNome(String nome) {
      this.nome = nome;
   }

   public String getSobrenome() {
      return sobrenome;
   }

   public void setSobrenome(String sobrenome) {
      this.sobrenome = sobrenome;
   }

   public boolean equals(Object obj) {
      Pessoa outro = (Pessoa) obj;
      if (nome.equals(outro.nome)) {
         if(sobrenome.equals(outro.sobrenome)) {
            return true;
         }
      }
      return false;
   }
}
```

Pronto, vemos que atende ao problema anterior e realiza corretamente a comparação para os dois objetos dados. Veja o resultado da execução na figura 5.

```
Pessoa@119c082
Pessoa@1add2dd
true
```

Figura 5. Execução de UtilizaPessoa.

Porém, há um problema. Na assinatura do método, temos que o método não deve lançar exceção de maneira alguma. Além disso, ele deve permitir receber qualquer classe para comparação (Object obj). O que aconteceria se classe Pessoa fosse comparada com outra classe qualquer? Altere a classe UtilizaPessoa para que fique como na listagem 7, e verifique o que ocorre.

Na execução da nova classe UtilizaPessoa teremos algo como na figura 6.

```
Pessoa@119c082
Xiiii
java.lang.ClassCastException
        at Pessoa.equals(Pessoa.java:22)
        at UtilizaPessoa.main(UtilizaPessoa.java:11)
Exception in thread "main"
```

Figura 6. Execução de UtilizaPessoa

Listagem 7. UtilizaPessoa.Java

```
public class UtilizaPessoa {
   public static void main (String [] args) {
      Pessoa fulano = new Pessoa();
      String objetoQualquer = "Xiiii";

      fulano.setNome("João");
      fulano.setSobrenome("Silva");

      System.out.println(fulano);
      System.out.println(objetoQualquer);
      System.out.println(
         fulano.equals(objetoQualquer));
   }
}
```

Verificamos um problema de conversão de tipo de objetos, então para evitar isso, devemos alterar nossa classe Pessoa, de modo que ela identifique os tipos de objetos e caso não seja o tipo desejado, retorne falso, sem lançar exceção.

Para solucionar, utilizaremos o operador instanceof que retorna verdadeiro ou falso, caso um objeto seja de uma classe ou não. Veja método equals na listagem 8.

Listagem 8. Pessoa.Java

```
public class Pessoa {
   private String nome;
   private String sobrenome;

   public String getNome() {
      return nome;
   }

   public void setNome(String nome) {
      this.nome = nome;
   }

   public String getSobrenome() {
      return sobrenome;
   }

   public void setSobrenome(String sobrenome) {
      this.sobrenome = sobrenome;
   }

   public boolean equals(Object obj) {
      if (obj instanceof Pessoa) {
         Pessoa outro = (Pessoa) obj;
         if (nome.equals(outro.nome)) {
            if(sobrenome.equals(outro.sobrenome)) {
               return true;
            }
         }
      }
      return false;
   }
}
```

A execução agora ocorre sem exceções e corretamente. Veja figura 7.

```
Pessoa@119c082
Xiiii
false
```

Figura 7. Execução de UtilizaPessoa.

Ótimo, agora não teremos mais problemas com objetos de outras classes, fora Pessoa. Estamos quase terminando.

O que aconteceria se um dos atributos da classe Pessoa fosse nulo? Vamos alterar a classe UtilizaPessoa, conforme listagem 9.

Listagem 9. UtilizaPessoa.Java

```
public class UtilizaPessoa {
    public static void main (String [] args) {
        Pessoa fulano = new Pessoa();
        Pessoa sicrano = new Pessoa();

        fulano.setNome("João");
        sicrano.setNome("João");

        System.out.println(fulano);
        System.out.println(sicrano);
        System.out.println(fulano.equals(sicrano));
    }
}
```

O resultado da execução seria algo como na figura 8.

```
Pessoa@119c082
Pessoa@1add2dd
java.lang.NullPointerException
        at Pessoa.equals(Pessoa.java:25)
        at UtilizaPessoa.main(UtilizaPessoa.java:12)
Exception in thread "main"
```

Figura 8. Execução de UtilizaPessoa

O programador recebrá uma exceção NullPointerException e o programa irá parar de funcionar. Não é muito bonito, mas facilmente contornável. Basta verificar, antes de comparar, se o atributo é nulo, e fazer as devidas correções, algo como:

```
if (nome==null) {
        if (outro.nome!=null) {
            return false;
        }
    } else { // nome não é nulo
        if (!nome.equals(outro.nome)) {
            return false;
        }
    }
return true;
```

Ou de maneira mais compacta:

```
( nome==null ? other.nome==null : nome.equals(outro.nome) )
```

Assim, a classe Pessoa irá ficar conforme listagem 10.

Listagem 10. Classe Pessoa.

```
public class Pessoa {
    private String nome;
    private String sobrenome;

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getSobrenome() {
        return sobrenome;
    }

    public void setSobrenome(String sobrenome) {
        this.sobrenome = sobrenome;
    }

    public boolean equals(Object obj) {
        if (obj instanceof Pessoa) {
            Pessoa outro = (Pessoa) obj;
            if ( (nome==null ? outro.nome==null : nome.equals(outro.nome)) ) {
                if( (sobrenome==null ? outro.sobrenome==null :
                        sobrenome.equals(outro.sobrenome)) ) {
                    return true;
                }
            }
        }
        return false;
    }
}
```

E o resultado da execução seria algo como na figura 9.

```
Pessoa@119c082
Pessoa@1add2dd
true
```

Figura 9. Execução de UtilizaPessoa.

Quase tudo certo, falta apenas um detalhe. Ao criar regras para considerar objetos iguais, devemos também alterar o método hashCode().

hashCode()

O código hash é uma representação de um objeto através de um inteiro e é utilizado para facilitar e otimizar o processo de armazenamento de um objeto em coleções e tabelas associativas. As tabelas associativas são um tipo especial de coleção na qual temos um objeto servindo como índice e associado a esse objeto temos outro, ou seja, dado um índice, obtemos o valor armazenado. São exemplos de tabelas associativas os mapas e tabelas hash. Então, deve-se esperar que objetos iguais tenham códigos hash iguais.

O padrão é calculado a partir do endereço da referência do objeto, mas para o nosso caso, esse comportamento resulta em algo indesejado. Antes de continuar a leitura do texto, analise a listagem 11.

Listagem 11. UtilizaPessoa.java

```
import java.util.HashMap;

public class UtilizaPessoa {
   public static void main (String [] args) {
      Pessoa fulano = new Pessoa();
      Pessoa sicrano = new Pessoa();

      fulano.setNome("João");
      sicrano.setNome("João");

      HashMap map = new HashMap();
      map.put(fulano, "Encontrei o Objeto");

      System.out.println(fulano);
      System.out.println(sicrano);
      System.out.println(fulano.equals(sicrano));
      System.out.println(fulano.hashCode());
      System.out.println(sicrano.hashCode());
      System.out.println(map.get(sicrano));
   }
}
```

A classe HashMap permite guardarmos um objeto associado a outro. No caso, guardamos a String "Encontrei o Objeto" associado ao objeto-chave, fulano. Assim, se quisermos resgatar o objeto guardado (String), basta entrar com a chave novamente (fulano).

Era de se esperar que pudéssemos usar o objeto sicrano, que é igual ao objeto fulano para obter o objeto guardado, mas isso não acontece devido ao problema do hashCode. Veja na figura 10 o resultado da execução.

```
Pessoa@119c082
Pessoa@1add2dd
true
18464898
28168925
null
```

Figura 10. Execução de UtilizaPessoa.

Para então obtermos o funcionamento correto, devemos escrever um método hashCode() condizente com o nosso método equals(), calculando a partir dos mesmos atributos utilizados para o cálculo de igualdade. O número obtido é um inteiro, que deve permanecer o mesmo a cada execução. Veja a listagem 12.

Listagem 12. Pessoa.Java

```
public class Pessoa {
   private String nome;
   private String sobrenome;

   public String getNome() {
      return nome;
   }
   public void setNome(String nome) {
      this.nome = nome;
   }
   public String getSobrenome() {
      return sobrenome;
   }
   public void setSobrenome(String sobrenome) {
      this.sobrenome = sobrenome;
   }

   public boolean equals(Object obj) {
      if (obj instanceof Pessoa) {
         Pessoa outro = (Pessoa) obj;
         if ( (nome==null ? outro.nome==null :
               nome.equals(outro.nome)) ) {
            if( (sobrenome==null ?
                 outro.sobrenome==null :
                sobrenome.equals(outro.sobrenome)) ) {
               return true;
            }
         }
      }
      return false;
   }

   public int hashCode() {
      int codigo = 37; // Semente
      if (nome!=null) {
         codigo = codigo*37 + nome.hashCode();
      }
      if (sobrenome!=null) {
         codigo = codigo*37 + sobrenome.hashCode();
      }
      return codigo;
   }
}
```

O que temos aqui é a criação de um número inteiro que representa o objeto dado, ou seja, um número que é associado ao objeto, que poderá ser usado como índice no lugar do objeto. Esse número é único e será sempre o mesmo, caso os atributos permaneçam o mesmo. Assim sendo, qualquer objeto Pessoa criado, que tiver o mesmo nome e mesmo sobrenome, possuirá igual número hash calculado, independentemente da hora em que esse objeto foi criado.

Partimos de um número qualquer, usaremos um número primo para obtermos números hash com menor possibilidade de repetição de valores. E, para cada atributo relevante (que foi usado no cálculo de igualdade), iremos realizar uma multiplicação do conteúdo anterior e somar o código hash calculado para a String de nome e para a String de sobrenome. No final, para dois pares iguais de nome e sobrenome, teremos um mesmo número hash calculado e, para pares distintos de nome e sobrenome, muito provavelmente teremos um número hash não-igual.

```
Pessoa@23656a
Pessoa@23656a
true
2319722
2319722
Encontrei o Objeto
```

Figura 10. Execução de UtilizaPessoa.

Temos agora o funcionamento adequado e correto dos objetos em questão.

Conclusão

Para um funcionamento adequado de um programa, é necessário conhecer certos detalhes da linguagem. Um deles é o uso dos métodos equals() e hashCode(). O desconhecimento de tais aspectos, resulta em problemas de funcionamento, perda de dados, além de resultados inesperados com relação à execução do programa. MJ