10 Coisas Que Eu Odeio Em Java

Rafael Santos

MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

www.lac.inpe.br/∼rafael.santos

Tópicos

1. Sobre...
2. Irritações com a linguagem
3. Frustrações com a API
4. Finalmente...

Objetivos

- Entender que nem tudo em Java é como parece!
- Apreciar que às vezes somente *achamos* que entendemos o que estamos fazendo.
- Eventualmente aprender a tomar cuidado com algumas armadilhas.
- Achar graça e coçar a cabeça!

Apresentação

- Veremos vários exemplos de código em Java que não se comportam como esperado.

Apresentação

- Veremos vários exemplos de código em Java que não se comportam como esperado.
- Em alguns casos, existem explicações...

Apresentação

- Veremos vários exemplos de código em Java que não se comportam como esperado.
- Em alguns casos, existem explicações...
- Boa parte dos comportamentos estranhos são documentados na *Java Language Specification*...

Apresentação

- Veremos vários exemplos de código em Java que não se comportam como esperado.
- Em alguns casos, existem explicações...
- Boa parte dos comportamentos estranhos são documentados na *Java Language Specification*...
- Mas são inesperados do mesmo jeito!

Apresentação

- Veremos vários exemplos de código em Java que não se comportam como esperado.
- Em alguns casos, existem explicações...
- Boa parte dos comportamentos estranhos são documentados na *Java Language Specification*...
- Mas são inesperados do mesmo jeito!
- É possível que você já conheça alguns destes problemas... mas e a explicação?

Como verificar?

- Sistema operacional: Ubuntu 8.04, kernel 2.6.24-24-generic
- IDE: Eclipse 3.4 com compilador javac 1.6.0_07 da Sun.
  - Erros diferentes compilando da IDE e da linha de comando!
  - É possível que algumas coisas sejam diferentes em outros ambientes...
  - Algumas inconsistências podem existir entre diferentes máquinas virtuais (veja as referências).
- **Importante:** algumas classes contém erros propositais que não permitem a compilação. Ao invés de comentar no código, usei marcadores.

Mentirinhas

Mentirinha 1: Evidentemente não odeio Java!

- Uso Java desde 1996.
- Autor de *Introdução à Programação Orientada a Objetos usando Java* (Campus/SBC).
- Mantenho o site *Java Image Processing Cookbook* (`http://www.lac.inpe.br/~rafael.santos/JIPCookbook/index.jsp`)
- Desenvolvo algoritmos de classificação e mineração de dados exclusivamente em Java.

Mentirinha 2: São mais que 10 coisas...

Casts inesperados

```
public class Cast
  {
  public static void main(String[] args)
    {
    float a = 1.299f; float b = 2.899f;
    int c = a+b; // cannot convert from float to int
    int d = 0;
    d += a; d += b;
    System.out.println(d); // 3!
    }
  }
```

Casts inesperados

```
public class Cast
  {
  public static void main(String[] args)
    {
    float a = 1.299f; float b = 2.899f;
    int c = a+b; // cannot convert from float to int
    int d = 0;
    d += a; d += b;
    System.out.println(d); // 3!
    }
  }
```

Explicação: `E1 op= E2` equivale a `E1 = (T) ((E1) op (E2))` onde `(T)` é o tipo de `E1`.

Ímpares negativos

```
public class Modulo
  {
  public static boolean éÍmpar(int x)
    {
    return ((x % 2) == 1);
    }
  public static void main(String[] args)
    {
    System.out.println(éÍmpar(0));  // false
    System.out.println(éÍmpar(1));  // true
    System.out.println(éÍmpar(2));  // false
    System.out.println(éÍmpar(-2)); // false
    System.out.println(éÍmpar(-1)); // false ??
    }
  }
```

Ímpares negativos

```
public class Modulo
  {
  public static boolean éÍmpar(int x)
    {
    return ((x % 2) == 1);
    }
  public static void main(String[] args)
    {
    System.out.println(éÍmpar(0));  // false
    System.out.println(éÍmpar(1));  // true
    System.out.println(éÍmpar(2));  // false
    System.out.println(éÍmpar(-2)); // false
    System.out.println(éÍmpar(-1)); // false ??
    }
  }
```

Explicação: Operador % retorna mesmo sinal do primeiro operando.

Infinitos

```
public class Infinitos
  {
  public static void main(String[] args)
    {
    System.out.println(Float.NEGATIVE_INFINITY+
                       Float.MAX_VALUE);          // -Infinity
    System.out.println(Float.NEGATIVE_INFINITY+
                       Float.POSITIVE_INFINITY); // NaN
    System.out.println(Float.POSITIVE_INFINITY+
                       Float.POSITIVE_INFINITY); // Infinity
    System.out.println(Float.POSITIVE_INFINITY-
                       Float.POSITIVE_INFINITY); // NaN
    }
  }
```

Infinitos

```
public class Infinitos
  {
  public static void main(String[] args)
    {
    System.out.println(Float.NEGATIVE_INFINITY+
                       Float.MAX_VALUE);          // -Infinity
    System.out.println(Float.NEGATIVE_INFINITY+
                       Float.POSITIVE_INFINITY); // NaN
    System.out.println(Float.POSITIVE_INFINITY+
                       Float.POSITIVE_INFINITY); // Infinity
    System.out.println(Float.POSITIVE_INFINITY-
                       Float.POSITIVE_INFINITY); // NaN
    }
  }
```

Explicação: `Float.POSITIVE_INFINITY`, `Float.NEGATIVE_INFINITY` são processados como símbolos.

Extremos

```
public class Extremos
  {
  public static void main(String[] args)
    {
    System.out.println(Integer.MAX_VALUE);   //  2147483647
    System.out.println(Integer.MIN_VALUE);   // -2147483648
    System.out.println(Integer.MAX_VALUE+1); // MIN_VALUE
    System.out.println(Integer.MIN_VALUE-1); // MAX_VALUE
    System.out.println(-Integer.MAX_VALUE);  // -2147483647: -MAX_VALUE
    System.out.println(-Integer.MIN_VALUE);  // -2147483648: MIN_VALUE
    }
  }
```

Extremos

```
public class Extremos
  {
  public static void main(String[] args)
    {
    System.out.println(Integer.MAX_VALUE);   //  2147483647
    System.out.println(Integer.MIN_VALUE);   // -2147483648
    System.out.println(Integer.MAX_VALUE+1); // MIN_VALUE
    System.out.println(Integer.MIN_VALUE-1); // MAX_VALUE
    System.out.println(-Integer.MAX_VALUE);  // -2147483647: -MAX_VALUE
    System.out.println(-Integer.MIN_VALUE);  // -2147483648: MIN_VALUE
    }
  }
```

Explicação: Por causa da representação interna, `Integer.MIN_VALUE == -Integer.MIN_VALUE`.

NaNs

```
public class NaN
  {
  public static void main(String[] args)
    {
    System.out.println(Float.NaN+3.4028235e38f); // NaN
    System.out.println(Float.NaN * 2);           // NaN
    System.out.println(Float.NaN-Float.NaN);     // NaN
    System.out.println(1f/Float.NaN);            // NaN
    System.out.println(Float.NaN/0);             // NaN
    System.out.println(Float.NaN/Float.NaN);     // NaN
    }
  }
```

NaNs

```
public class NaN
  {
  public static void main(String[] args)
    {
    System.out.println(Float.NaN+3.4028235e38f); // NaN
    System.out.println(Float.NaN * 2);           // NaN
    System.out.println(Float.NaN-Float.NaN);     // NaN
    System.out.println(1f/Float.NaN);            // NaN
    System.out.println(Float.NaN/0);             // NaN
    System.out.println(Float.NaN/Float.NaN);     // NaN
    }
  }
```

Explicação: `Float.NaN` são processados como símbolos (com outras regras).

Mais NaNs

```
public class NaN2
  {
  public static void main(String[] args)
    {
    float[] f = {Float.NEGATIVE_INFINITY,Float.MAX_VALUE,
                 Float.MIN_VALUE,Float.NaN,
                 Float.POSITIVE_INFINITY,1f,2f,3f};
    float menor = Float.NaN;
    for(float valor:f)
      if (valor < menor) menor = valor;
    System.out.println(menor); // NaN
    }
  }
```

Mais NaNs

```
public class NaN2
  {
  public static void main(String[] args)
    {
    float[] f = {Float.NEGATIVE_INFINITY,Float.MAX_VALUE,
                 Float.MIN_VALUE,Float.NaN,
                 Float.POSITIVE_INFINITY,1f,2f,3f};
    float menor = Float.NaN;
    for(float valor:f)
      if (valor < menor) menor = valor;
    System.out.println(menor); // NaN
    }
  }
```

Explicação: `Float.NaN` e outros valores não são comparáveis.

Ainda mais NaNs

```
public class NaN3
  {
  public static void main(String[] args)
    {
    float fip = Float.POSITIVE_INFINITY;
    float fin = Float.NEGATIVE_INFINITY;
    float fnan = Float.NaN;
    double dip = Double.POSITIVE_INFINITY;
    double din = Double.NEGATIVE_INFINITY;
    double dnan = Double.NaN;
    System.out.println(fip == dip);   // true
    System.out.println(fin == din);   // true
    System.out.println(fnan == dnan); // false
    System.out.println(fnan == fnan); // false
    }
  }
```

Ainda mais NaNs

```
public class NaN3
  {
  public static void main(String[] args)
    {
    float fip = Float.POSITIVE_INFINITY;
    float fin = Float.NEGATIVE_INFINITY;
    float fnan = Float.NaN;
    double dip = Double.POSITIVE_INFINITY;
    double din = Double.NEGATIVE_INFINITY;
    double dnan = Double.NaN;
    System.out.println(fip == dip);   // true
    System.out.println(fin == din);   // true
    System.out.println(fnan == dnan); // false
    System.out.println(fnan == fnan); // false
    }
  }
```

Explicação: Infinitos são símbolos equivalentes, *NaN*s não são iguais a nada (nem a eles mesmos!)

Precisão

```
public class Precisao
  {
  public static void main(String[] args)
    {
    System.out.println(2.0-1.7); // 0.30000000000000004
    System.out.println((2.0-1.7) == (0.3)); // false
    System.out.println(1-((1/7)+(2/7)+(4/7))); // 1!
    System.out.println(1-((1./7.)+(2./7.)+(4./7.))); // 0, ok.
    }
  }
```

Precisão

```
public class Precisao
  {
  public static void main(String[] args)
    {
    System.out.println(2.0-1.7); // 0.30000000000000004
    System.out.println((2.0-1.7) == (0.3)); // false
    System.out.println(1-((1/7)+(2/7)+(4/7))); // 1!
    System.out.println(1-((1./7.)+(2./7.)+(4./7.))); // 0, ok.
    }
  }
```

Explicações: Aritmética de ponto flutuante **não é precisa**.

Precisão

```
public class Precisao
  {
  public static void main(String[] args)
    {
    System.out.println(2.0-1.7); // 0.30000000000000004
    System.out.println((2.0-1.7) == (0.3)); // false
    System.out.println(1-((1/7)+(2/7)+(4/7))); // 1!
    System.out.println(1-((1./7.)+(2./7.)+(4./7.))); // 0, ok.
    }
  }
```

Explicações: Aritmética de ponto flutuante **não é precisa**. Números em divisões **não são** automaticamente `float` ou `double`.

Mais precisão

```
public class Precisao2
  {  // 1e-10f * 100.000.000.000 = 10
  public static void main(String[] args)
    {
    float muitoPequeno = 1e-10f; float temp;
    // 100 * 1.000.000.000
    temp = 0f;
    for(int i=0;i<100;i++) temp += (1000000000f*muitoPequeno);
    System.out.println(temp); // 10.000002
    // 100.000 * 1.000.000
    temp = 0f;
    for(int i=0;i<100000;i++) temp += (1000000f*muitoPequeno);
    System.out.println(temp); // 10.00631
    // 100.000.000 * 1.000
    temp = 0f;
    for(int i=0;i<100000000;i++) temp += (1000f*muitoPequeno);
    System.out.println(temp); // 2.0
    }
  }
```

Mais precisão

```
public class Precisao2
  {  // 1e-10f * 100.000.000.000 = 10
  public static void main(String[] args)
    {
    float muitoPequeno = 1e-10f; float temp;
    // 100 * 1.000.000.000
    temp = 0f;
    for(int i=0;i<100;i++) temp += (1000000000f*muitoPequeno);
    System.out.println(temp); // 10.000002
    // 100.000 * 1.000.000
    temp = 0f;
    for(int i=0;i<100000;i++) temp += (1000000f*muitoPequeno);
    System.out.println(temp); // 10.00631
    // 100.000.000 * 1.000
    temp = 0f;
    for(int i=0;i<100000000;i++) temp += (1000f*muitoPequeno);
    System.out.println(temp); // 2.0
    }
  }
```

Explicações: perda de precisão cumulativa??

Overflow

```
public class Overflow
  {
  public static void main(String[] args)
    {
    long microsPorHora = 60 * 60 * 1000 * 1000;
    long millisPorHora = 60 * 60 * 1000;
    System.out.println(microsPorHora / millisPorHora); // -193
    microsPorHora = 60L * 60 * 1000 * 1000;
    millisPorHora = 60L * 60 * 1000;
    System.out.println(microsPorHora / millisPorHora); // 1000
    float muitosAnosEmMillis =
       24*365*250000*millisPorHora; // 7.8839998E15
    System.out.println(muitosAnosEmMillis); // -7.5778825E15
    }
  }
```

Overflow

```
public class Overflow
  {
  public static void main(String[] args)
    {
    long microsPorHora = 60 * 60 * 1000 * 1000;
    long millisPorHora = 60 * 60 * 1000;
    System.out.println(microsPorHora / millisPorHora); // -193
    microsPorHora = 60L * 60 * 1000 * 1000;
    millisPorHora = 60L * 60 * 1000;
    System.out.println(microsPorHora / millisPorHora); // 1000
    float muitosAnosEmMillis =
       24*365*250000*millisPorHora; // 7.8839998E15
    System.out.println(muitosAnosEmMillis); // -7.5778825E15
    }
  }
```

Explicação: operações são em `ints`, existe perda de precisão antes da atribuição a um `long` ou `float`.

Autoboxing

```
public class Autoboxing
  {
  public static void main(String[] args)
    {
    float fn = 123; // ok
    Float fi1 = 123; // cannot convert from int to Float
    Float fi2 = fn;  // ok
    Float fi3 = new Float(123);   // ok
    Float fi4 = "123"; // cannot convert from String to Float
    Float fi5 = new Float("123"); // ok
    Float fi6 = 123.0; // cannot convert from double to Float
    Float fi7 = new Float(123.0); // ok
    Double d = 123.456;
    Float fi8 = d; // cannot convert from Double to Float
    Float fi9 = new Float(d); // ok
    }
  }
```

Autoboxing

```
public class Autoboxing
  {
  public static void main(String[] args)
    {
    float fn = 123; // ok
    Float fi1 = 123; // cannot convert from int to Float
    Float fi2 = fn;  // ok
    Float fi3 = new Float(123);   // ok
    Float fi4 = "123"; // cannot convert from String to Float
    Float fi5 = new Float("123"); // ok
    Float fi6 = 123.0; // cannot convert from double to Float
    Float fi7 = new Float(123.0); // ok
    Double d = 123.456;
    Float fi8 = d; // cannot convert from Double to Float
    Float fi9 = new Float(d); // ok
    }
  }
```

Explicação: uso direto de *autoboxing* depende da existência de um construtor adequado.

Mais *Autoboxing*

```
public class Autoboxing2
  {
  private static double soma(Object[] obs)
    {
    double soma = 0;
    for(Object o:obs)
      {
      soma += o; // operator += undefined for double, Object
      soma += (Double)o; // ClassCastException
      }
    return soma;
    }
  public static void main(String[] args)
    {
    Object[] objetos1 = {123,456.78,'d'};
    System.out.println(soma(objetos1));
    }
  }
```

Mais *Autoboxing*

```
public class Autoboxing2
  {
  private static double soma(Object[] obs)
    {
    double soma = 0;
    for(Object o:obs)
      {
      soma += o; // operator += undefined for double, Object
      soma += (Double)o; // ClassCastException
      }
    return soma;
    }
  public static void main(String[] args)
    {
    Object[] objetos1 = {123,456.78,'d'};
    System.out.println(soma(objetos1));
    }
  }
```

Explicação: podemos fazer *autoboxing* **para** um `Object` mas não **de** um `Object`.

Métodos e atributos estáticos

```
public class ContadorEstatico
  {
  private static int contador=0;
  public static int próximo()
    {
    contador++;
    return contador;
    }
  }
```

```
public class UsaContador
  {
  public static void main(String[] args)
    {
    System.out.println(ContadorEstatico.próximo()); // 1
    System.out.println(ContadorEstatico.próximo()); // 2
    System.out.println(ContadorEstatico.próximo()); // 3
    }
  }
```

Métodos e atributos estáticos

```
public class ContadorEstatico
  {
  private static int contador=0;
  public static int próximo()
    {
    contador++;
    return contador;
    }
  }
```

```
public class UsaContador
  {
  public static void main(String[] args)
    {
    System.out.println(ContadorEstatico.próximo()); // 1
    System.out.println(ContadorEstatico.próximo()); // 2
    System.out.println(ContadorEstatico.próximo()); // 3
    }
  }
```

Até aqui tudo bem...

Métodos e atributos estáticos

```
public class ContadorCarros extends ContadorEstatico
  {
  public static void contaCarro()
    {
    System.out.println(próximo()+" carros");
    }
  }
```

```
public class ContadorBicicletas extends ContadorEstatico
  {
  public static void contaBicicletas()
    {
    System.out.println(próximo()+" bicicletas");
    }
  }
```

Métodos e atributos estáticos

```
public class ContadorCarros extends ContadorEstatico
  {
  public static void contaCarro()
    {
    System.out.println(próximo()+" carros");
    }
  }
```

```
public class ContadorBicicletas extends ContadorEstatico
  {
  public static void contaBicicletas()
    {
    System.out.println(próximo()+" bicicletas");
    }
  }
```

Até aqui tudo bem...

Métodos e atributos estáticos

```
public class UsaContadores
  {
  public static void main(String[] args)
    {
    ContadorCarros.contaCarro();           // 1 carros
    ContadorBicicletas.contaBicicletas(); // 2 bicicletas
    ContadorCarros.contaCarro();           // 3 carros
    ContadorBicicletas.contaBicicletas(); // 4 bicicletas
    ContadorBicicletas.contaBicicletas(); // 5 bicicletas
    }
  }
```

Métodos e atributos estáticos

```
public class UsaContadores
  {
  public static void main(String[] args)
    {
    ContadorCarros.contaCarro();           // 1 carros
    ContadorBicicletas.contaBicicletas(); // 2 bicicletas
    ContadorCarros.contaCarro();           // 3 carros
    ContadorBicicletas.contaBicicletas(); // 4 bicicletas
    ContadorBicicletas.contaBicicletas(); // 5 bicicletas
    }
  }
```

Explicação: somente uma cópia dos campos estáticos é compartilhada pelas classes herdeiras.

Sobrecarga de métodos

```
public class Sobrecarga1
  {
  public Sobrecarga1(Object o) { System.out.println("Object"); }
  public Sobrecarga1(double[] d) { System.out.println("double[]"); }
  public Sobrecarga1(Double d) { System.out.println("Double"); }
  public Sobrecarga1(Date d) { System.out.println("Date"); }
  public Sobrecarga1(double d) { System.out.println("double"); }
  }
```

```
public class UsaSobrecarga1
  {
  public static void main(String[] args)
    {
    new Sobrecarga1((Double)null); // Double
    new Sobrecarga1((Integer)null); // Object
    new Sobrecarga1((double[])null); // double[]
    new Sobrecarga1(null); // the constructor Sobrecarga1(Object)
    }                      // is ambiguous!
  }
```

Sobrecarga de métodos

```
public class Sobrecarga1
  {
  public Sobrecarga1(Object o) { System.out.println("Object"); }
  public Sobrecarga1(double[] d) { System.out.println("double[]"); }
  public Sobrecarga1(Double d) { System.out.println("Double"); }
  public Sobrecarga1(Date d) { System.out.println("Date"); }
  public Sobrecarga1(double d) { System.out.println("double"); }
  }
```

```
public class UsaSobrecarga1
  {
  public static void main(String[] args)
    {
    new Sobrecarga1((Double)null); // Double
    new Sobrecarga1((Integer)null); // Object
    new Sobrecarga1((double[])null); // double[]
    new Sobrecarga1(null); // the constructor Sobrecarga1(Object)
    }                      // is ambiguous!
  }
```

Explicação: `null` pode ser `Object` ou `Double` ou `Date`!

Mais sobrecarga de métodos

```
public class Sobrecarga2
  {
  public Sobrecarga2(Object o) { System.out.println("Object"); }
  public Sobrecarga2(double[] d) { System.out.println("double[]"); }
  public Sobrecarga2(double d) { System.out.println("double"); }
  } // sem ambiguidades?
```

```
public class UsaSobrecarga2
  {
  public static void main(String[] args)
    {
    new Sobrecarga2((Double)null);   // Object
    new Sobrecarga2((Integer)null);  // Object
    new Sobrecarga2((double[])null); // double[]
    new Sobrecarga2(null);           // double[]
    }
  }
```

Mais sobrecarga de métodos

```
public class Sobrecarga2
  {
  public Sobrecarga2(Object o) { System.out.println("Object"); }
  public Sobrecarga2(double[] d) { System.out.println("double[]"); }
  public Sobrecarga2(double d) { System.out.println("double"); }
  } // sem ambiguidades?
```

```
public class UsaSobrecarga2
  {
  public static void main(String[] args)
    {
    new Sobrecarga2((Double)null);    // Object
    new Sobrecarga2((Integer)null);   // Object
    new Sobrecarga2((double[])null); // double[]
    new Sobrecarga2(null);            // double[]
    }
  }
```

Explicação: `double[]` é mais específico que `Object`.

Ancestrais e herdeiras

```
public class Ancestral
  {
  public String meuNome = "Ancestral";
  public String getNome() { return meuNome; }
  }
```

```
public class Herdeira extends Ancestral
  {
  private String meuNome = "Herdeira";
  public String getNome() { return meuNome; }
  private String getNome() { return meuNome; } // Cannot reduce
                                                // visibility
  }
```

Ancestrais e herdeiras

```
public class Ancestral
  {
  public String meuNome = "Ancestral";
  public String getNome() { return meuNome; }
  }
```

```
public class Herdeira extends Ancestral
  {
  private String meuNome = "Herdeira";
  public String getNome() { return meuNome; }
  private String getNome() { return meuNome; } // Cannot reduce
                                               // visibility
  }
```

Até aqui tudo bem...

Campos e atributos final

```
public class Ancestral2
  {
  public static final String meuNome = "Ancestral2";
  public static final String getStatic() { return meuNome; }
  public final String getNome() { return meuNome; }
  }
```

```
public class Herdeira2 extends Ancestral2
  {
  public static final String meuNome = "Herdeira2";
  public static final String getStatic()// cannot override final method
    { return meuNome; }
  public final String getNome() // cannot override final method
    { return meuNome; }
  }
```

Campos e atributos final

```
public class Ancestral2
  {
  public static final String meuNome = "Ancestral2";
  public static final String getStatic() { return meuNome; }
  public final String getNome() { return meuNome; }
  }
```

```
public class Herdeira2 extends Ancestral2
  {
  public static final String meuNome = "Herdeira2";
  public static final String getStatic()// cannot override final method
    { return meuNome; }
  public final String getNome() // cannot override final method
    { return meuNome; }
  }
```

Até aqui tudo bem...

Campos e atributos final

```
public class UsaHerdeira2
  {
  public static void main(String[] args)
    {
    System.out.println(Herdeira2.meuNome); // Herdeira2
    System.out.println(Herdeira2.getStatic()); // Ancestral2
    System.out.println(new Herdeira2().getNome()); // Ancestral2
    }
  }
```

Campos e atributos `final`

```
public class UsaHerdeira2
  {
  public static void main(String[] args)
    {
    System.out.println(Herdeira2.meuNome); // Herdeira2
    System.out.println(Herdeira2.getStatic()); // Ancestral2
    System.out.println(new Herdeira2().getNome()); // Ancestral2
    }
  }
```

Explicações: Significado de `final` é diferente para atributos e métodos!

Campos e atributos `final`

```
public class UsaHerdeira2
  {
  public static void main(String[] args)
    {
    System.out.println(Herdeira2.meuNome); // Herdeira2
    System.out.println(Herdeira2.getStatic()); // Ancestral2
    System.out.println(new Herdeira2().getNome()); // Ancestral2
    }
  }
```

Explicações: Significado de `final` é diferente para atributos e métodos!
Método da classe ancestral ignora atributo da herdeira.

Exceções e finally

```
public class Excecoes1
  {
  public static int exec()
    {
    int res = 0;
    try
      { res = 1; System.out.println("Res: "+res); }
    catch (Exception e)
      { res = 2; System.out.println("Res: "+res); }
    finally
      { res = 3; System.out.println("Res: "+res); }
    return res;
    }
  public static void main(String[] args)
    {
    System.out.println(exec()); // Res: 1;  Res: 3;  3
    }
  }
```

Exceções e finally

```
public class Excecoes1
  {
  public static int exec()
    {
    int res = 0;
    try
      { res = 1; System.out.println("Res: "+res); }
    catch (Exception e)
      { res = 2; System.out.println("Res: "+res); }
    finally
      { res = 3; System.out.println("Res: "+res); }
    return res;
    }
  public static void main(String[] args)
    {
    System.out.println(exec()); // Res: 1;  Res: 3;  3
    }
  }
```

Até aqui tudo bem...

Mais exceções e finally

```
public class Excecoes2
  {
  public static int exec()
    {
    try
      { return 1; }
    catch (Exception e)
      { return 2; }
    finally // Eclipse: finally block does not complete normally
      { return 3; }
    }
  public static void main(String[] args)
    {
    System.out.println(exec()); // 3
    }
  }
```

Mais exceções e finally

```
public class Excecoes2
  {
  public static int exec()
    {
    try
      { return 1; }
    catch (Exception e)
      { return 2; }
    finally // Eclipse: finally block does not complete normally
      { return 3; }
    }
  public static void main(String[] args)
    {
    System.out.println(exec()); // 3
    }
  }
```

Até aqui tudo bem... (apesar do *warning*): **regra da terminação súbita** (`finally` sempre executado).

Ainda mais exceções e finally

```
public class Excecoes3
  {
  public static int exec()
    {
    int res = 0;
    try
      { res = 1; System.out.println("Res: "+res); return res; }
    catch (Exception e)
      { res = 2; System.out.println("Res: "+res); }
    finally
      { res = 3; System.out.println("Res: "+res); }
    return res;
    }
  public static void main(String[] args)
    {
    System.out.println(exec()); //  Res: 1;  Res: 3;  1
    }
  }
```

Ainda mais exceções e finally

```
public class Excecoes3
  {
  public static int exec()
    {
    int res = 0;
    try
      { res = 1; System.out.println("Res: "+res); return res; }
    catch (Exception e)
      { res = 2; System.out.println("Res: "+res); }
    finally
      { res = 3; System.out.println("Res: "+res); }
    return res;
    }
  public static void main(String[] args)
    {
    System.out.println(exec()); // Res: 1; Res: 3; 1
    }
  }
```

O return ocorre no bloco try.

Ainda mais exceções e finally

```
public class Excecoes3
  {
  public static int exec()
    {
    int res = 0;
    try
      { res = 1; System.out.println("Res: "+res); return res; }
    catch (Exception e)
      { res = 2; System.out.println("Res: "+res); }
    finally
      { res = 3; System.out.println("Res: "+res); }
    return res;
    }
  public static void main(String[] args)
    {
    System.out.println(exec()); //  Res: 1;  Res: 3;  1
    }
  }
```

O return ocorre no bloco try.

Por que a modificação do valor foi revertida?

Exceções e finally (final!)

```
public class Excecoes4
  {
  public static int exec()
    {
    int res = 0;
    try
      { res = 1; System.out.println("Res: "+res); System.exit(1); }
    catch (Exception e)
      { res = 2; System.out.println("Res: "+res); }
    finally
      { res = 3; System.out.println("Res: "+res); }
    return res;
    }
  public static void main(String[] args)
    {
    System.out.println(exec()); // Res: 1
    }
  }
```

Exceções e finally (final!)

```
public class Excecoes4
  {
  public static int exec()
    {
    int res = 0;
    try
      { res = 1; System.out.println("Res: "+res); System.exit(1); }
    catch (Exception e)
      { res = 2; System.out.println("Res: "+res); }
    finally
      { res = 3; System.out.println("Res: "+res); }
    return res;
    }
  public static void main(String[] args)
    {
    System.out.println(exec()); // Res: 1
    }
  }
```

E a regra da terminação súbita?

Exceções e finally (final!)

```
public class Excecoes4
  {
  public static int exec()
    {
    int res = 0;
    try
      { res = 1; System.out.println("Res: "+res); System.exit(1); }
    catch (Exception e)
      { res = 2; System.out.println("Res: "+res); }
    finally
      { res = 3; System.out.println("Res: "+res); }
    return res;
    }
  public static void main(String[] args)
    {
    System.out.println(exec()); // Res: 1
    }
  }
```

E a regra da terminação súbita?

Explicação: OK, `System.exit()` é ainda mais súbito!

Instâncias imutáveis

```
public class InstanciasImutaveis
  {
  public static void main(String[] args)
    {
    String texto = "10 coisas que eu odeio em ";
    texto.concat("Visual ");
    texto += "Fortran";
    System.out.println(texto); // 10 coisas que eu odeio em Fortran
    BigInteger i = new BigInteger("1000");
    i.add(new BigInteger("2000"));
    i.add(new BigInteger("4000"));
    System.out.println(i); // 1000
    }
  }
```

Instâncias imutáveis

```
public class InstanciasImutaveis
  {
  public static void main(String[] args)
    {
    String texto = "10 coisas que eu odeio em ";
    texto.concat("Visual ");
    texto += "Fortran";
    System.out.println(texto); // 10 coisas que eu odeio em Fortran
    BigInteger i = new BigInteger("1000");
    i.add(new BigInteger("2000"));
    i.add(new BigInteger("4000"));
    System.out.println(i); // 1000
    }
  }
```

Explicação: Instâncias de `String`, `BigInteger` e *wrappers* são imutáveis.

Datas inválidas

```
public class DataErrada
  {
  public static void main(String[] args)
    {
    Calendar c = Calendar.getInstance();
    c.clear(); c.setLenient(false);
    c.set(Calendar.DAY_OF_MONTH,21);
    c.set(Calendar.MONTH,11);
    c.set(Calendar.YEAR,1945);
    System.out.println(c.getTime()); // Fri Dec 21 00:00:00 BRT 1945
    c.set(Calendar.DAY_OF_MONTH,132);
    System.out.println(c.getTime()); // IllegalArgumentException:
    }                                // DAY_OF_MONTH;
```

Datas inválidas

```
public class DataErrada
  {
  public static void main(String[] args)
    {
    Calendar c = Calendar.getInstance();
    c.clear(); c.setLenient(false);
    c.set(Calendar.DAY_OF_MONTH,21);
    c.set(Calendar.MONTH,11);
    c.set(Calendar.YEAR,1945);
    System.out.println(c.getTime()); // Fri Dec 21 00:00:00 BRT 1945
    c.set(Calendar.DAY_OF_MONTH,132);
    System.out.println(c.getTime()); // IllegalArgumentException:
    }                                // DAY_OF_MONTH;
```

Valor do mês é entre 0 e 11.

Datas inválidas

```
public class DataErrada
  {
  public static void main(String[] args)
    {
    Calendar c = Calendar.getInstance();
    c.clear(); c.setLenient(false);
    c.set(Calendar.DAY_OF_MONTH,21);
    c.set(Calendar.MONTH,11);
    c.set(Calendar.YEAR,1945);
    System.out.println(c.getTime()); // Fri Dec 21 00:00:00 BRT 1945
    c.set(Calendar.DAY_OF_MONTH,132);
    System.out.println(c.getTime()); // IllegalArgumentException:
    }                                // DAY_OF_MONTH;
```

Valor do mês é entre 0 e 11.

Explicação: Leniência só é verificada por alguns métodos.

Buffer

```
public class Buffer
  {
  public static void main(String[] args)
    {
    String mensagem = "Leia-me com atenção!";
    for (int i=0;i<mensagem.length();i++)
      System.out.write(mensagem.charAt(i));
    }
  }
```

Buffer

```
public class Buffer
  {
  public static void main(String[] args)
    {
    String mensagem = "Leia-me com atenção!";
    for (int i=0;i<mensagem.length();i++)
      System.out.write(mensagem.charAt(i));
    }
  }
```

Nada é impresso, mesmo com a finalização do programa.

Buffer

```
public class Buffer
  {
  public static void main(String[] args)
    {
    String mensagem = "Leia-me com atenção!";
    for (int i=0;i<mensagem.length();i++)
      System.out.write(mensagem.charAt(i));
    }
  }
```

Nada é impresso, mesmo com a finalização do programa.

Explicação: `write` não faz `flush` no *stream*.

Algumas observações (para programadores Java)

- Nem tudo está perdido:
  - Muitos dos exemplos são artificiais ou extremos.
  - Não devem ocorrer no dia-a-dia.
- Não deu para ver muitos exemplos...
  - ... foco foi nos que afetam ensino de Java como linguagem de programação.
  - Ainda existem muitas outras pequenas confusões!
- Veja as referências para mais exemplos e casos estranhos.

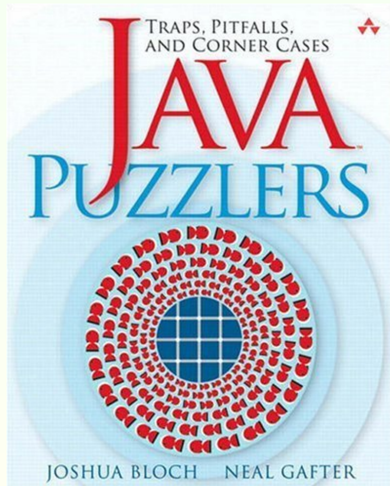
Lições (para programadores Java)

- Matemática com valores extremos é diferente. Cuidado em especial com laços.
- Instâncias de algumas classes são imutáveis.
- Testar software é barato. Escreva testes para casos extremos.
- Não confie somente no que a documentação da API diz. Teste para ver se você entendeu mesmo.
- **Simplifique seu código!**
- Simplifique a organização de suas classes e pacotes!

Referências

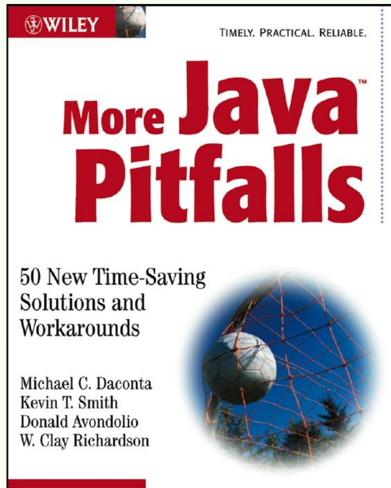
Pouco depois que começei a colecionar estas esquisitices, *Joshua Bloch* e *Neal Gafter* lançaram o livro *Java Puzzlers: Traps, Pitfalls, and Corner Cases* (Addison-Wesley, 304pp., Junho 2005).

O livro contém exemplos e explicações para alguns dos problemas mencionados nesta palestra e muitos outros.



Referências

Ainda outro livro interessante, *More Java Pitfalls: 50 New Time-Saving Solutions and Workarounds*, de *Michael C. Daconta*, *Kevin T. Smith*, *Donald Avondolio* e *W. Clay Richardson* (Wiley, 476pp., 2003) mostra várias armadilhas e pegadinhas.



Referências

Alguns problemas interessantes e complexos mostrados no *More Java Pitfalls*:

- Vários problemas relacionados com `Process proc = rt.exec("cmd");`
- Dicas para programação de interfaces gráficas.
- Dicas sobre entrada/saída.
- *Parsing* de documentos XML.
- **Dicas para aplicações Web.**

Referências

Jien-Tsai Chen, *Wuu Yang* e *Jing-Wei Huang* demonstraram em *Traps in Java* (The Journal of Systems and Software 72, pp. 33-47, 2004) que nem todas as implementações se comportam da mesma forma!

- Mostraram também que é possível escrever código realmente não-intuitivo.

Veja também a *Java Language Specification, Third Edition* em `http://java.sun.com/docs/books/jls/index.html`.

- Divirta-se! Veja a seção 15.12.2.5 *(Choosing the Most Specific Method)*.

Programadores em VB achando graça?

Nota: estas informações são meio antigas mas ainda interessantes.

- `Dim I As Integer` *versus* `Dim I, J, K, L As Integer`: J K e L são **variantes**!
- O próprio conceito de variantes!
- Índices de arrays começam com zero ou um?
- Chamada de funções: `F(3)`. Elementos de arrays: `F(3)`.
- Inicialização de arrays: `Dim A(20) As Double`, `A(0) = 4.5`, `A(1) = 4.71`, `A(2) = 4.82`, `A(3) = 4.92`... e nada de arrays de constantes!

Programadores em VB achando graça?

- `Integer` de 16 bits!
- `Subname param1, param2` → OK.
  `R=Funcname(param1, param2)` → OK.
  `Subname(param1, param2)` → erro de sintaxe.
- Aparentes diferenças entre `Funcname(param1, param2)` e `CALL Funcname(param1, param2)` e passagem de parâmetros por valor ou referência.

Isso foi resolvido com incompatibilização de versões – veja as referências!

Programadores em VB achando graça?

Referências:

- *Thirteen ways to loathe VB*: http://www.ddj.com/documents/s=1503/ddj0001vs/jan00.htm
- *Visual Basic .not*: http://vb.mvps.org/vfred/
- *Developers cry foul over new Microsoft language*: http://news.com.com/2009-1001-251154.html?legacy=cnet

Eu uso uma linguagem SUPERIOR!

- Padrões: **C++98**, **C++03** e em breve **C++0x**.
- Inferência de tipos:

```
auto otherVariable = 5;
auto itr = myvec.begin();
auto c = 0; decltype(c) e;
```

- Funções lambda: `[](int x,int y) { return x+y; }`

Veja isto e muito mais na Wikipedia (http://en.wikipedia.org/wiki/C++0x)

Eu uso uma linguagem AINDA MELHOR!

- Encadeamento de *namespaces* (*nested namespaces*).
- `String == string`?
- *Operator overloading* pode ser usado para o bem e para o mal: `blah+argh`?
- `goto`!
- Classes parciais!!!
- Lado bom: ao menos `float f[] = new float[10]` não compila.

Um artigo interessante:
http://www.25hoursaday.com/CsharpVsJava.html

Comentários finais

- É possível criticar qualquer linguagem, embora seja mais fácil criticar algumas do que outras.
- A melhor é a que a gente conhece bem.
- Idiomas e idiossincrasias não devem ser base para decisões finais.
- Vivemos em tempos interessantes: centenas de linguagens com diversas características (Wikipedia: *Alphabetical list of programming languages*). Quais usar? Quais evitar?