Programação em Java com J2SE 5.0

Atualização: J2SE 5.0

Helder da Rocha

Julho 2005

Objetivos

- Este módulo explora as principais novidades do J2SE 5.0 (exceto genéricos e utilitários de concorrência*)
  1. Enhanced for loop
  2. Autoboxing e unboxing
  3. Typesafe enumerations
  4. Varargs
  5. Static import
  6. Metadata (annotations)
  7. Formatação de texto

* abordados em módulos à parte

For que itera sobre coleções

- Enhanced for loop (JLS 14.14.2)
- A instrução for agora aceita uma sintaxe alternativa

```
for (Objeto obj: colecao) { ... }
```

  - Pode ser lido como: “repita o bloco para cada Objeto obj de coleção”
- Pode ser usado com arrays ou objetos iteráveis (que implementem a nova interface **Iterable**)
  - **`Iterable`** tem método **`Iterator<E> iterator()`**
  - Todas as coleções, filas, mapas, etc. implementam **`Iterable`** (que é genérica: **`Iterable<E>`**).

Exemplos

- Coleções (Iterable)

```
Collection<String> sc =
   new ArrayList<String>();
sc.add("Hello");
sc.add(" ");
sc.add("World");
String result = "";
```

– Usando for convencional

```
for (Iterator<String> i =
     sc.iterator(); i.hasNext(); )
    result += i.next();
System.out.println(result);
```

– Usando enhanced for

```
for (String i: sc) {
    result += i;
}
System.out.println(result);
```

- Arrays

```
int[] a = {1,2,3,4,5,6,7,8,9,10};
int sum = 0;
```

– Usando for convencional

```
for (int i = 0; i < a.length; i++)
    sum += a[i];
System.out.println(sum);
```

– Usando enhanced for

```
for (int i : a)
    sum += i;
System.out.println(sum);
```

As expressões acima são equivalentes!

Loops aninhados

- Ficam muito mais simples com o novo **for**
- O código abaixo tem um erro. Qual é?

```
List<Suit> suits = ...; List<Rank> ranks = ...
List<Card> sortedDeck = new ArrayList<Card>();
for (Iterator<Suit> i = suits.iterator(); i.hasNext(); ) {
    for (Iterator<Rank> j = ranks.iterator(); j.hasNext(); )
        sortedDeck.add(new Card(i.next(), j.next()));
}
```

Causa NoSuchElementException

- É difícil de achar! Tem chamadas next() demais

```
for (Iterator<Suit> i = suits.iterator(); i.hasNext(); ) {
    Suit suit = i.next(); // consertado! next() fora do segundo loop!
    for (Iterator<Rank> j = ranks.iterator(); j.hasNext(); )
        sortedDeck.add(new Card(suit, j.next()));
}
```

- Com o novo **for**, fica bem mais simples

```
for (Suit suit : suits)
    for (Rank rank : ranks)
        sortedDeck.add(new Card(suit, rank));
```

Conversão automática de tipos

- Autoboxing e Unboxing (Type Conversions JLS 5.1.7 e 5.1.8)
  - 8 valores primitivos são empacotados automaticamente em objetos e vice-versa
  - Aplica-se a todas as atribuições e expressões
  - Facilita o uso de tipos primitivos em coleções
- Antes era preciso fazer

```
Map products = new HashMap();
products.put(new Integer(2779), new Float(20.45f));
products.put(Integer.valueOf(922), Float.valueOf(99.99f));
float price = new Float( (Float) products.get(
                new Integer(922).intValue()) ).floatValue();
```

- Agora pode-se fazer

```
Map<Integer, Float> products = new HashMap<Integer, Float>();
products.put(2779, 20.45f);
products.put(922, 99.99f);
float price = products.get(922);
```

autoboxing

unboxing (retorno do método)

Exemplo

- Um adaptador que implementa a interface List e converte arrays de inteiros

```
public static List<Integer> asList(final int[] a) {
    return new AbstractList<Integer>() {
        public Integer get(int i) { return a[i]; }
        // Throws NullPointerException if val == null
        public Integer set(int i, Integer val) {
            Integer oldVal = a[i];
            a[i] = val;
            return oldVal;
        }
        public int size() { return a.length; }
    };
}
```

boxing

boxing

unboxing

- Muito conciso, porém baixa performance
  - Boxing e unboxing impacta performance
  - Usar objetos quando poderia-se usar primitivos consome mais recursos desnecessariamente
  - Trade-off: legibilidade vs. performance

Resumo: quando usar?

- For-each
  - Use sempre que possível
  - Não pode ser usado quando precisa-se ter acesso ao iterador. Ex: remover elementos ao atravessar a coleção, modificar o elemento atual, iterar sobre múltiplas coleções
  - Ao escrever uma API, implemente a interface Iterable nos objetos que deverão ser passados para o for-each
- Autoboxing
  - Quando houver “descasamento de impedâncias” entre tipos primitivos e objetos empacotadores
  - Não abuse: um Integer não substitui um int (a performance é muito, muito pior)

Enums: anti-padrão

- A solução padrão para representar uma enumeração era

```
public static final int INVERNO = 0;
public static final int PRIMAVERA = 1;
public static final int VERAO = 2;
public static final int OUTONO= 3;
```

int enum anti-pattern

- As constantes geralmente eram usadas em métodos, como por exemplo

```
setEstacao(int estacao) { ... }
```

Esse padrão tem vários problemas

- 1. Não é typesafe: como estacao é apenas um int, o seguinte é possível:
  `setEstacao(4);` estação não existente
  `setEstacao(INVERNO + VERAO);` forma estranha de produzir um outono
- 2. Não tem namespace
  - Mistura-se com outras constantes existentes na classe
  - Pode haver colisões de nome com constantes herdadas
- 3. Requer recompilação do código se novas constantes forem adicionadas
- 4. Traz nenhuma informação útil
  - Os valores que as constantes contém são apenas ints

Typesafe enums: pré-Java 5

- Antes dos Enums, a solução era implementar typeface enumerations (Joshua Bloch, *Effective Java*, item 21)

```
public class Suit implements Comparable {
    private final String name;
    private static int nextOrdinal = 0;
    private final int ordinal = nextOrdinal++;
    private Suit(String name) { this.name = name; }
    public String toString() { return name; }
    public int compareTo(Object obj) {
        return ordinal - ((Suit)o).ordinal;
    }
    public static final Suit CLUBS = new Suit("clubs");
    public static final Suit DIAMONDS = new Suit("diamonds");
    public static final Suit HEARTS = new Suit("hearts");
    public static final Suit SPADES = new Suit("spades");
}
```

- Desvantagem: código complexo (consequentemente mais vulnerável, desestimula o uso contra anti-pattern)

Typesafe enums

(Enums JLS 8.9)

- Antes

```
public static final int INVERNO = 0;
public static final int PRIMAVERA = 1;
public static final int VERAO = 2;
public static final int OUTONO= 3;
...
void setEstacao(int estacao) { ... }
```

- Depois

```
enum Estacao {INVERNO, PRIMAVERA, VERAO, OUTONO};
...
void setEstacao(Estacao estacao) { ... }
```

- Tem mais: enums são classes!
  - Podem ter construtores, métodos, etc.

Typesafe enums: exemplo

```
public class Card {
    public enum Rank { DEUCE, THREE, FOUR, FIVE, SIX,
        SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE }

    public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }

    private final Rank rank;
    private final Suit suit;
    private Card(Rank rank, Suit suit) {
        this.rank = rank;
        this.suit = suit;
    }

    public String toString() { return rank + " of " + suit; }

    private static final List<Card> protoDeck = new ArrayList<Card>();

    // Initialize prototype deck
    static {
        for (Suit suit : Suit.values())
            for (Rank rank : Rank.values())
                protoDeck.add(new Card(rank, suit));
    }
}
```

Declaração similar à de uma classe interna
(o "**;**" não é necessário)

Veja este mesmo exemplo usando estratégia antiga no livro Effective Java, item 21

Typesafe enums: exemplo (2)

```
public enum Planet {
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6);

    private final double mass;   // in kilograms
    private final double radius; // in meters
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
    private double mass()   { return mass; }
    private double radius() { return radius; }

    // universal gravitational constant  (m3 kg-1 s-2)
    public static final double G = 6.67300E-11;

    double gravity() {
        return G * mass / (radius * radius);
    }
    double weight(double otherMass) {
        return otherMass * surfaceGravity();
    }
}
```

top level

constantes

este construtor será chamado para inicializar as constantes

Para usar (exemplos):

```
Planet p =
 Planet.EARTH;
int g =
 Planet.EARTH.gravity();
```

Enums: comportamento variável

- É possível atribuir comportamentos diferentes a constantes
  - **Constant-specific methods**: método abstrato é implementado na chamada de cada constante

```
public enum Operation {
  PLUS   { double eval(double x, double y) { return x + y; } },
  MINUS  { double eval(double x, double y) { return x - y; } },
  TIMES  { double eval(double x, double y) { return x * y; } },
  DIVIDE { double eval(double x, double y) { return x / y; } };

  // Do arithmetic op represented by this constant
  abstract double eval(double x, double y);
}
```

Veja este mesmo exemplo usando estratégia antiga no livro Effective Java, item 21

Varargs: argumentos variáveis

(Method declarations JLS 8.4.1)

- Permite a passagem de um número previamente indeterminado de argumentos nos métodos
  - Antes era preciso ter uma quantidade fixa, e passar um array
  - Hoje, o sistema aceita o array ou seus argumentos “soltos”
- Regras
  - Tem que ser o último argumento declarado no método
  - Argumentos têm que ser de mesmo tipo ou tipo conversível (equivalente a um array)
- Exemplo de sintaxe:
  - declaração

```
void metodo (char op, String arg2, double... args) {}
```

  - uso

```
obj.metodo(‘s’, “Soma”, 13, 3L, 4.5f, 1,2,3,4,5);
```

Exemplos de varargs

- Antes fazia-se assim
  - Declaração do método

```
public static String format(String pattern, Object[] args);
```

  - Uso

```
Object[] args = {new Integer(7), new Date(), "a disturbance"};
String result = MessageFormat.format(
  "At {1,time} on {1,date}, there was {2} on planet "
  + "{0,number,integer}.", args);
```

- Agora pode-se fazer assim
  - Declaração do método

```
public static String format(String pattern, Object... args);
```

  - Uso

```
String result = MessageFormat.format(
  "At {1,time} on {1,date}, there was {2} on planet "
  + "{0,number,integer}.", 7, new Date(), "a disturbance");
```

autoboxing

args

Importação de constantes estáticas

(Static imports: JLS 7.5.3 e 7.5.4)

- Antes

```
String dateString =
    now.get(Calendar.HOUR) +
    now.get(Calendar.MINUTE);
```

- Depois
  - Antes de sua classe:

```
import static java.util.Calendar.*;
```

  - Dentro do método

```
String dateString =
    now.get(HOUR) +
    now.get(MINUTE);
```

TODAS as constantes de Calendar foram importadas!

- Use com cuidado! Não abuse!
  - Importação de constantes estáticas (static import) misturam o namespace da sua classe com outros namespaces: pode causar conflitos e obter o resultado oposto (que é maior legibilidade)

Quando usar: varags, enums, ...

- Varargs
  - Não abuse
  - Em APIs use apenas quando houver um benefício real
  - Pode dificultar e tornar mais complexo o reuso (sobrecarga e sobreposição)
- Enums
  - Use sempre que precisar!
- Static imports
  - Não abuse (pode trazer resultado oposto)
  - Use apenas quando quiser cair na tentação de usar soluções piores, como herdar de interfaces de constantes (anti-pattern)

Annotations (metadados)

(Annotation types JLS 9.6, Annotations JLS 9.7)

- “Oficialização” do uso de doclets para geração de código (ex: XDoclet)
  - `/** @tag-especial arg=valor */`
- Meta-informações ignoradas pelo compilador que podem ser inseridas no meio do código para uso por outras ferramentas
  - Podem aparecer em qualquer lugar
  - Podem ser simples marcadores ou conter elementos que podem receber valores
  - Diminuem a necessidade de se ter arquivos (XML) separados para configuração
  - Podem tornar o código muito ilegível: dependem de ferramentas que possam ocultar as anotações

Annotations

- O recurso de metadados em Java 5.0 consiste de
  - Uma sintaxe para a declaração de metadados (annotation types)
  - Uma sintaxe para uso de metadados
  - Uma API para ler metadados e uma classe para representar metadados
  - Uma ferramenta (apt) para processamento básico
- Tipicamente um programador-cliente irá
  - Inserir metadados cujo vocabulário é definido por alguma ferramenta que está usando (ex: JAX-RPC, Hibernate, EJB 3.0)
  - Rodar sua(s) ferramentas ou usar seu framework (eles irão ler os metadados e fazer algo com eles)
- Programadores de ferramentas/frameworks irão
  - Definir novos tipos de metadados

Benefícios

- Redução do espalhamento de código em frameworks
- Exemplo: servidor JAX-RPC.
  - Antes:

```
public interface HelloIF extends Remote {
    public String sayHello(String s) throws RemoteException;
}
public class HelloImpl implements HelloIF {
    public String sayHello(String s) {
        return “Hello “+s;
    }
}
```

  - Depois:

```
public class HelloImpl {
    public @remote String HelloImpl(String s) {
        return "Hello “+s;
    }
}
```

@remote é compreendido pelas ferramentas do JAX-RPC que gera automaticamente a interface necessária

Veja também: EJB 3.0: única classe vs. 2 interfaces, bean e XML do EJB 2.0

Declaração de metadados

- Um programador que está criando uma ferramenta ou framework, pode criar novos metadados
- A declaração é semelhante à declaração de interfaces
  - A principal diferença é o @ antes da palavra interface

```
public @interface RequestForEnhancement {
    int    id();
    String synopsis();
    String engineer() default "[unassigned]";
}
```

- Para usar, o programador-cliente chama a anotação pelo nome (precedida de @) em algum lugar do código

```
@RequestForEnhancement(
    id       = 2868724,
    synopsis = "Enable time-travel",
    engineer = "Mr. Peabody“ )
public static void travelThroughTime(Date destination) { ... }
```

Compilador vai ignorar tudo
Processador pode usar dados

Marcadores e valores

- Uso comum de metadados (fora dos grandes frameworks) geralmente é mais simples
- Marcadores
  - Definição

```
public @interface Importante {}
```

  - Uso

```
@Importante public class Tarefa { ...}
```

- Valores
  - Quando há apenas um valor, deve ser value()

```
public @interface Copyright { String value(); }
```

  - Ele pode ser omitido na chamada

```
@Copyright("2002 Yoyodyne Propulsion Systems")
public class OscillationOverthruster { ... }
```

Transformação de texto

- Formatter
  - Permite formatar (e imprimir) em estilo C
  - Utilitário em **System.out**: método printf()

```
System.out.printf("Local time: %tT", Calendar.getInstance());
System.err.printf("Unable to open file '%1$s': %2$s",
                  fileName, exception.getMessage());
```

  - Uso em String semelhante a **sprintf()** do C

```
Calendar c = new GregorianCalendar(1995, MAY, 23);
String s = String.format("Duke's Birthday: %1$tm %1$te,%1$tY", c);
```

  - Criação de Formatter

```
StringBuilder sb = new StringBuilder();
Formatter formatter = new Formatter(sb, Locale.US);
formatter.format("%2$2s %1$2s", "a", "b"); // -> " b  a"
formatter.format(“R$ %(,.2f", saldo); // -> “R$ (1,442.00)"
```

- Scanner
  - Faz o inverso do formatter (leitura formatada)

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
```

Outros recursos úteis

- ProcessBuilder
  - Builder design pattern
  - Mais interação com processos externos à JVM: Runtime.exec()

```
ProcessBuilder pb =
   new ProcessBuilder("myCommand", "myArg1", "myArg2");
Map<String, String> env = pb.environment();
env.put("VAR1", "myValue");
env.remove("OTHERVAR");
env.put("VAR2", env.get("VAR1") + "suffix");
pb.directory("myDir");
Process p = pb.start();
```

Constrói o processo

Inicia o processo

- String System.getenv(String)
  - Antes deprecated, este método permite ler variáveis de ambiente do sistema diretamente

```
String variavel = System.getenv("PATH");
```

Veja também

- StringBuilder
  - Igual a StringBuffer, só que não é thread-safe
  - Mais eficiente em ambientes não-concorrentes
- Recursos de gerenciamento
  - java.lang.instrument
  - java.lang.management (JMX)
- Uma grande API de recursos para concorrência
  - java.util.concurrent
  - Novas coleções: Queue
- Documentação Java
  - docs/relnotes/features.html
  - Java Language Specification (capítulos em vermelho)

Fontes de pesquisa

[1] Documentação do J2SDK 1.5

[2] Joshua Bloch, Effective Java, Addison-Wesley, 2001.

- Este livro é muito bom e vai demorar para ficar completamente obsoleto; Veja Item 21: typesafe enum (versão antiga)

[3] James Gosling, Bill Joy, Guy Steele e Gilad Bracha, "The Java Language Specification, 3rd Edition". Addison-Wesley, 2005 (disponível em java.sun.com).

- Diversas seções (veja indicações nos slides correspondentes em vermelho)

[4] David Flanagan. Five favorite features from five. O'Reilly – OnJava, 04/2005.

- www.onjava.com/pub/a/onjava/2005/04/20/javaIAN5.html
- Leia esta artigo: os features não são os usuais

[5] Diversas palestras e laboratórios do JavaONE

- Podem ser baixados em java.sun.com/javaone