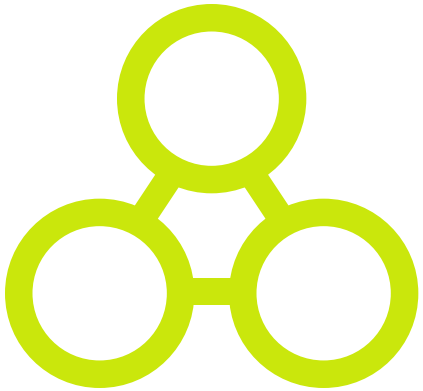


nemo

ontology & conceptual modeling research group

Desenvolvimento OO com Java

Conteúdo extra

Vítor E. Silva Souza

(vitor.souza@ufes.br)

http://www.inf.ufes.br/~vitorsouza

Departamento de Informática

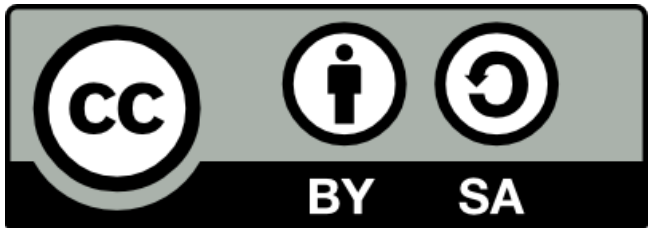
Centro Tecnológico

Universidade Federal do Espírito Santo

Licença para uso e distribuição

- Este obra está licenciada com uma licença Creative Commons Atribuição-CompartilhaIgual 4.0 Internacional;
- Você tem o direito de:
  - Compartilhar: copiar e redistribuir o material em qualquer suporte ou formato
  - Adaptar: remixar, transformar, e criar a partir do material para qualquer fim, mesmo que comercial.
- De acordo com os termos seguintes:
  - Atribuição: você deve dar o crédito apropriado, prover um link para a licença e indicar se mudanças foram feitas. Você deve fazê-lo em qualquer circunstância razoável, mas de maneira alguma que sugira ao licenciante a apoiar você ou o seu uso;
  - CompartilhaIgual: se você remixar, transformar, ou criar a partir do material, tem de distribuir as suas contribuições sob a mesma licença que o original.

Mais informações podem ser encontradas em:
http://creativecommons.org/licenses/by-sa/4.0/



Desenvolvimento OO com Java – Conteúdo extra

CARACTERÍSTICAS AVANÇADAS

Parâmetros variáveis

- A partir do Java 5 se tornou possível definir métodos que recebem um número variável de argumentos (*varargs*):

```
public class Teste {
  void print(boolean msg, String ... objs) {
     if (msg) System.out.println("Args:");
     for (int i = 0; i < objs.length; i++)
        System.out.println(objs[i]);
  }

  public static void main(String[] args) {
     Teste t = new Teste();
     t.print(true, "Java", "Sun", "JCP");
  }
}
```

Parâmetros variáveis

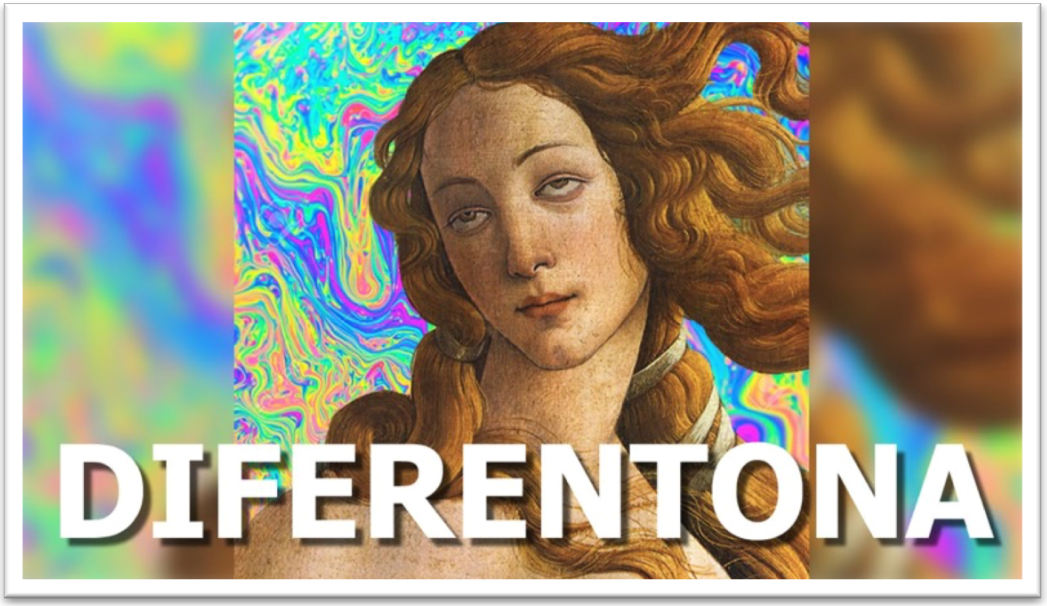
- Só pode haver uma lista de parâmetros variáveis na declaração do método;
- Deve ser a última a ser declarada;
- Funciona como um vetor do tipo declarado (no exemplo, vetor de `String`);
- Não há limite para o número de parâmetros;
- Também aceita zero parâmetros.

```
t.print(false, "A", "B", "C", "D", "E");
t.print(true, "Um", "Dois");
t.print(false);
```

Parâmetros variáveis

- O método main() agora pode ser assim então:

```
public class Teste {
  public static void main(String ... args) {
    // ...
  }
}
```



Blocos de inicialização estática

- No exemplo da classe `Aleatorio`, inicializamos uma variável no construtor porque não conseguíamos fazê-lo em uma só linha;
- E se esta variável for `static`?

```
class Aleatorio {
  int numero;

  Aleatorio(int max) {
     Random rand = new Random();
     numero = rand.nextInt(max);
  }
}
```

Blocos de inicialização estática

- Resolvemos a questão com blocos de inicialização estática;
- Os blocos estáticos de uma classe são executados quando a classe é usada pela 1ª vez.

```
class Aleatorio {
  static int numero;

  static {
     Random rand = new Random();
     numero = rand.nextInt(20);
  }
}
```

Blocos de inicialização não estática

- Também podemos fazer blocos de inicialização não-estática;
- Funcionam como os construtores: chamados em cada criação de objeto.

```
class Aleatorio {
  int numero;

  {
     Random rand = new Random();
     numero = rand.nextInt(20);
  }
}
```

Desenvolvimento OO com Java – Conteúdo extra

CRIANDO TIPOS GENÉRICOS

Tipos genéricos

- Novidade do Java 5.0;
- Funcionalidade já existente em outras linguagens (ex.: templates do C++);
- Teoria estudada e solidificada;
- Muitas bibliotecas são genéricas:
- Código complicado de ler e manter;
- Coerção leva a erros em tempo de execução.

Funcionamento

- Cria-se uma classe “genérica”:
  - Trabalha com um tipo T, desconhecido;
  - Tipo será atribuído na definição da referência.

```
public class Casulo<T> {
  private T elemento;

  public void colocar(T elem) {
     elemento = elem;
  }

  public T retirar() {
     return elemento;
  }
}
```

Funcionamento

- Referência e construtor definem o tipo manipulado pela classe genérica;
- Compilador pode efetuar checagens de tipo.

```
Casulo<String> cs = new Casulo<String>();
cs.colocar("Uma string");
// Erro: cs.colocar(new Integer(10));
String s = cs.retirar();

Casulo<Object> co = new Casulo<Object>();
co.colocar("Uma string");
co.colocar(new Integer(10));
Object o = co.retirar();
```

Herança de tipos genéricos

- Os conceitos de herança podem se confundir quando usamos tipos genéricos.

```
Casulo<String> cs = new Casulo<String>();
Casulo<Object> co = cs;
```

- O código acima gera erro;
- Por que? `Object` não é superclasse de `String`?

```
co.colocar(new Integer());  // OK!
String s = cs.retirar();    // OK!
```

- co e cs são o mesmo objeto e o código acima faria s receber um `Integer`!

Coringas (*wildcards*)

- Considere, então, um código genérico:

```
void imprimir(Casulo<Object> c) {
  System.out.println(c.retirar());
}
```

- O código não é tão genérico assim:

```
imprimir(co);   // OK!
imprimir(cs);   // Erro!
```

- Como acabamos de ver, não se pode converter `Casulo<String>` para `Casulo<Object>`!

Coringas (*wildcards*)

- Para essa situação, podemos usar coringas:

```
void imprimir(Casulo<?> c) {
  System.out.println(c.retirar());
}
```

- Significa: o método `imprimir()` pode receber casulos de qualquer tipo.

Coringas limitados

- Podemos também limitar o tipo genérico como sendo subclasse de alguma classe;

```
void desenhar(Casulo<? extends Forma> c) {
  c.retirar().desenhar();
  c.retirar().inverter();
}
```

- Significa: o método `imprimir()` pode receber casulos de qualquer subclasse de Forma ou casulos de Forma;
- O compilador garante que o que retirarmos do casulo será uma Forma ou uma subclasse.

Coringas limitados

- Porém, não podemos alterar uma classe com um coringa:

```
void teste(Casulo<? extends Forma> c) {
  // Erro: c pode ser Casulo<Retangulo>!
  c.colocar(new Circulo());
}

void outroTeste(Casulo<?> c) {
  // Erro: c pode ser Casulo<Integer>!
  c.colocar("Uma string!");
}

void desenhar(Casulo<? extends Forma> c) {
  c.retirar().desenhar();
  c.retirar().inverter();
}
```

Métodos genéricos

- Novamente, este método não é tão genérico:

```
static void arrayToCollection(Object[] array,
                              Collection<Object> collection) {
    for (Object o : array) collection.add(o);
}
```

- E este gera erro de compilação:

```
static void arrayToCollection(Object[] array,
                              Collection<?> collection) {
    for (Object o : array) collection.add(o); // Erro!
}
```

Métodos genéricos

- A solução:

```
static <T> void arrayToCollection(T[] array,
                                  Collection<T> collection) {
    for (T o : array) collection.add(o);
}
```

- O método usa um tipo diferente a cada chamada:

```
// usa Long[] e Collection<Long>:
arrayToCollection(new Long[] {1L, 2L},
                  new ArrayList<Long>());
```

Código legado

- Ao misturar código novo com código pré-genéricos, podem ocorrer problemas:

```
public class Teste {
  static Collection listaInt() {
     List l = new ArrayList();
     l.add(1); l.add(2); l.add(3);
     return l;
  }

  public static void main(String[] args) {
     Collection<Integer> lista = listaInt();
  }
}
```

Código legado

- Ao contrário do que se pode imaginar, `Collection` ≠ `Collection<Object>`;
- Classes sem definição do tipo genérico podem ser convertidas para classes com definição de qualquer tipo;
- Gera um aviso (*warning*) e o desenvolvedor deve assegurar que não haverá erro de execução.

Conclusões

- Usar tipos genéricos é relativamente simples e traz grandes vantagens;
- Criar tipos genéricos é mais complexo e envolve um entendimento mais aprofundado.

Desenvolvimento OO com Java – Conteúdo extra

CLASSES INTERNAS

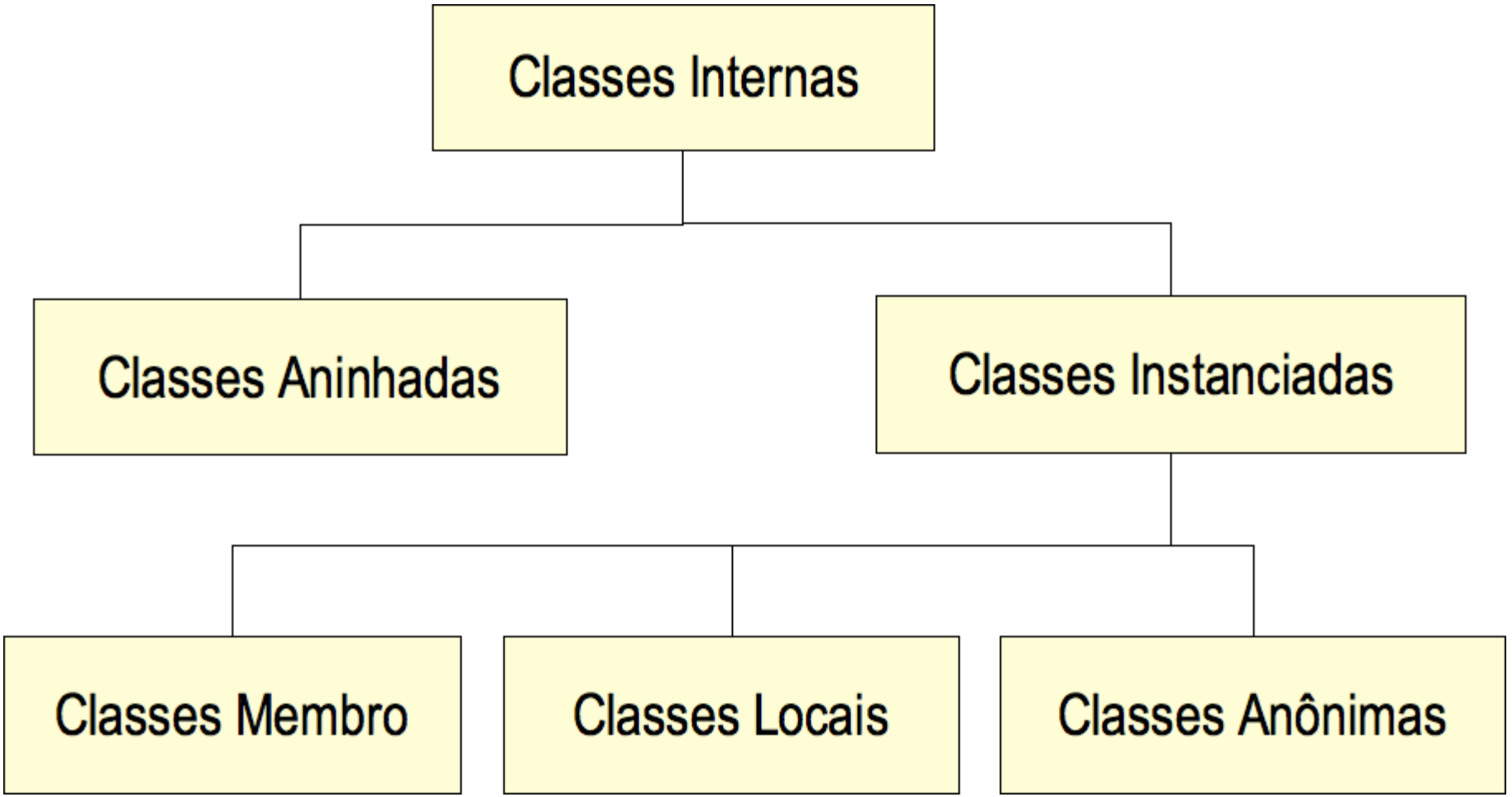
Classes externas e internas

- Até agora definimos classes no nível superior, dentro de pacotes;
- Podemos definir classes dentro de classes, como se fossem atributos ou métodos;
- Vantagens:
  - Legibilidade: agrupamento por similaridade;
  - Ocultamento: podem ser privadas ou protegidas;
  - Redigibilidade: classes internas possuem acesso aos membros privados da classe que a definiu e vice-versa. De fato, classes internas surgiram na versão 1.1 de Java com este propósito.

Código fonte e *bytecode*

- Como são definidas dentro de outras classes, ficam no mesmo arquivo `.java`;
- Ao compilar, gera-se vários arquivos `.class`, compondo o nome da classe externa e interna;
- Ex.:
  - `Externa.java` contém a classe Externa, que define uma classe interna chamada Interna;
  - Ao compilar, gera-se `Externa.class` e `Externa$Interna.class`.
- A classe interna não pode ter o mesmo nome da classe externa que a define.

Tipos de classes internas



Classes aninhadas

- Tipo mais simples de classe interna;
- Classe definida dentro de outra, mas funciona como classe de nível superior:
  - Definida como se fosse um membro `static`;
  - Referência via `Externa.Interna`, como se fosse um atributo estático;
- Permite definir acesso privado ou protegido e agrupa classes logicamente relacionadas.

Classes aninhadas

```
class Par {
  private Chave chave;
  private Valor valor;

  public Par(Chave chave, Valor valor) {
     this.chave = chave;
     this.valor = valor;
  }

  static class Chave {
     private String nome;

     public Chave(String nome) {
        this.nome = nome;
     }
  }

  // Classe Par continua...
```

Classes aninhadas

```
  protected static class Valor {
     private int valor;
     public Valor(int valor) {
        this.valor = valor;
     }
  }
}

public class Teste {
  public static void main(String[] args) {
     Par.Chave chave = new Par.Chave("Nota");
     Par.Valor valor = new Par.Valor(10);
     Par par = new Par(chave, valor);
  }
}
```

Classes instanciadas

- Ao contrário das classes aninhadas, classes instanciadas não são `static`;
- Por isso, é preciso ter um objeto da classe externa para referenciarmos a classe interna;
- Podem ser de três tipos:
  - Classes membro;
  - Classes locais;
  - Classes anônimas.

Classes membro

- Definida como um membro (não-estático) da classe;
- Portanto, pode ser pública, privada, protegida ou privativa ao pacote;
- Tem acesso total a todos os membros (inclusive privados) da classe que a define;
- Para ser criada, é preciso ter uma instância da classe que a define:
  - Referência: `Externa.Interna`;
  - Criação: `ext.new Interna();`

Classes membro

```
class TimeFutebol {
  private Tecnico tecnico;

  private Jogador[] time = new Jogador[23];

  class Pessoa {
     String nome;

     Pessoa(String nome) {
        this.nome = nome;
     }
  }

  private class Tecnico extends Pessoa {
     Tecnico(String nome) {
        super(nome);
     }
  }
                                    // Continua...
```

Classes membro

```
public class Jogador extends Pessoa {
   public Jogador(String nome) {
      super(nome);
   }
}

public void setTecnico(String nome) {
   this.tecnico = new Tecnico(nome);
}

public void addJogador(int c, Jogador j) {
   time[c] = j;
}
}                                          // Continua...
```

Classes membro

```
public class Teste {
  public static void main(String[] args) {
    TimeFutebol selecao = new TimeFutebol();
    selecao.setTecnico("Felipão");

    TimeFutebol.Jogador jogador;
    jogador = selecao.new Jogador("Júlio César");
    selecao.addJogador(12, jogador);

    jogador = selecao.new Jogador("Fred");
    selecao.addJogador(9, jogador);

    // ...
  }
}
```

Classes membro

- Bastante úteis quando precisamos criar classes que são utilizadas apenas por uma classe;
- Exemplos:
  - Uma lista encadeada e cada nó da lista;
  - Uma janela (GUI) e seus *listeners*;
  - Etc.
- Criam uma nova forma de utilizar `this`:
- `ClasseExterna.this.membro`.

Uso de this

```
class A {
  public String nome = "a";
  public String sobrenome = "s";
  public class B {
     public String nome = "b";
     public class C {
        public String nome = "c";
        public void imprime(String nome) {
           System.out.println(nome);
           System.out.println(this.nome);
           System.out.println(C.this.nome);
           System.out.println(B.this.nome);
           System.out.println(A.this.nome);
           System.out.println(sobrenome);
        }
     }
  }
}
```

Uso de this

```
public class Instanciadas {
  public static void main(String[] args) {
     A a = new A();
     A.B b = a.new B();
     A.B.C c = b.new C();
     c.imprime("d");
  }
}

// Resultado: d, c, c, b, a, s
```

Classes membro e interfaces

- Estratégia para esconder a classe interna e ainda assim utilizá-la externamente;
- Define-se a classe membro como `private` ou `protected`;
- Define-se sua interface como interface de nível superior (`public` ou *package-private*);
- Faz a classe interna implementar a interface;
- Adiciona um método na classe externa para retornar uma instância da classe interna, com *upcasting* para a interface.

Classes membro e interfaces

```
interface Bicho {
  String getNome();
}

class PetShop {
  private class Gato implements Bicho {
     String nome;

     public Gato(String nome) {
        this.nome = nome;
     }

     public String getNome() {
        return nome;
     }
  }

   // Classe PetShop continua...
```

Classes membro e interfaces

```
  public Bicho comprar(String nome) {
     return new Gato(nome);
  }
}

public class Teste {
  public static void main(String[] args) {
     PetShop loja = new PetShop();
     // Erro: PetShop.Gato bicho;
     Bicho bicho = loja.comprar("Garfield");
     System.out.println(bicho.getNome());
  }
}
```

Classes locais

- Classes declaradas como variáveis locais: dentro de blocos;
- Diferenças com relação à classes membro:
  - Assim como variáveis locais, não podem ter especificador de acesso;
  - Escopo de visibilidade: só podem ser usadas dentro do bloco nas quais foram definidas;
  - Acesso: possuem acesso às variáveis e parâmetros do bloco, desde que sejam declarados como final.
- Pode ser usada a mesma estratégia com interface explicada para classes membro.

Classes locais

```
public class Teste {
  static void f(final double d, long l) {
    final String s = "Pode";

    class Impressora {
      void imprimir() {
        System.out.println(d);
        System.out.println(s);
        // Erro: System.out.println(l);
        // Erro: System.out.println(r);
      }
    }

    final String r = "Não pode";
    Impressora imp = new Impressora();
    imp.imprimir();
  }

  // Classe Teste continua...
```

Classes locais

```
  public static void main(String[] args) {
     f(1.5, 1000);

     // Erro: Impressora imp2;
     // Erro: imp2 = new Impressora();
  }
}
```

Classes anônimas

- Semelhante às classes locais:
  - Mesmas regras (de escopo, de acesso, ...);
  - Não possuem nome;
  - Não podem definir construtores;
  - Só permitem a criação de uma instância.

Classes anônimas

```
interface Bicho {
  String getNome();
}

class PetShop {
  public Bicho comprar(final String nome) {
    return new Bicho() {
       public String getNome() {
          return nome;
       }
    };      // Note o ";"
  }
}

// Continua...
```

Classes anônimas

```
public class Teste {
  public static void main(String[] args) {
     PetShop loja = new PetShop();
     Bicho bicho = loja.comprar("Odie");
     System.out.println(bicho.getNome());
  }
}
```

Sintaxe complexa

- Classes anônimas são criadas a partir de outras classes ou interfaces;
- Sintaxe estranha à primeira vista, porém promove grande redigibilidade;
  - A adição em complexidade é compensada pela facilidade na escrita de códigos muito comuns, principalmente na construção de interfaces gráficas.
- É como se criássemos uma classe local que estendesse uma outra ou implementasse uma interface.

Código equivalente com classe local

```
class PetShop {
  public Bicho comprar(final String nome) {
    class BichoAnonimo implements Bicho {
       public String getNome() {
          return nome;
       }
    }
    return new BichoAnonimo();
  }
}
```

Nomes dos arquivos de bytecode

- Como classes anônimas não possuem nome, Java usa números inteiros para os nomes dos arquivos `.class`:
  - `PetShop.class` contém a classe `PetShop`;
  - `PetShop$1.class` contém a classe anônima.

Níveis de aninhamento

- Classes aninhadas podem, por sua vez, definir classes internas aninhadas;
- `Par.Chave.Subchave`, etc.;
- Classes instanciadas (internas e não-aninhadas) não podem definir classes aninhadas;
- Classes instanciadas podem definir classes internas também instanciadas.

Quando usar?

- Classes Aninhadas:
  - Agrupar classes logicamente relacionadas.
- Classes Membro:
  - Esconder uma classe que é utilizada somente dentro de uma outra classe.
- Classes Locais:
  - Quando o escopo é um bloco e são criadas várias instâncias.
- Classes Anônimas:
  - Quando o escopo é um bloco e é criada apenas uma instância.

Desenvolvimento OO com Java – Conteúdo extra

NOVIDADES DO JAVA 8

Novidades do Java 8

- Veremos exemplos rápidos de:
  - Expressões lambda;
  - Referências a métodos;
  - Métodos default;
  - API de stream;
  - Optional;
  - Nova API de datas/horas.

Fontes

- O mínimo que você deve saber de Java 8: http://blog.caelum.com.br/o-minimo-que-voce-deve-saber-de-java-8/
- Conheça a nova API de datas do Java 8: http://blog.caelum.com.br/conheca-a-nova-api-de-datas-do-java-8/
- Streams e datas para os desafios do dia a dia no Java 8: http://blog.caelum.com.br/streams-e-datas-para-os-desafios-do-dia-a-dia-no-java-8/
- What's New in Java 8: https://leanpub.com/whatsnewinjava8/read
- Java Optional Objects: http://java.dzone.com/articles/java-optional-objects

Java Magazine, edição 62 (agosto 2008)

SEÇÃO JAVA: NESTA SEÇÃO VOCÊ ENCONTRA ARTIGOS INTERMEDIÁRIOS E AVANÇADOS SOBRE JAVA

Closures em Java

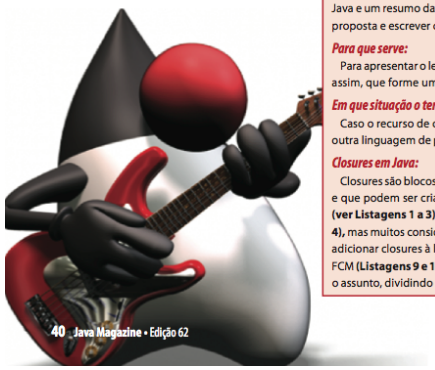
Propostas de novo recurso trazem muita dis

Closure é um conceito de linguagens de programação proveniente das linguagens funcionais que também é presente em muitas linguagens orientadas a objetos (e, portanto, imperativas), como Smalltalk, Groovy, Ruby, Python e Scala.

Nos últimos dois anos se discute a adição de closures à linguagem Java. Dado o grande impacto que isso teria na linguagem – por uns julgado positivo, por outros negativo – muito se tem discutido nas conferências, blogs e portais sobre Java se o recurso deveria ser realmente adicionado e, em caso positivo, qual seria a melhor forma de fazê-lo.

Tanta discussão gerou três propostas principais para adição de closures, conhecidas pelas siglas CICE+ARM, FCM e BGGA. Uma delas, a BGGA, já é disponível experimentalmente em builds do OpenJDK. Neste artigo, apresentaremos o que são closures, as três propostas para sua adição à linguagem Java e um resumo da polêmica em torno do assunto.

Ainda que a disponibilidade de uma nova versão do Java incorporando closures seja um plano futuro e incerto, esperamos que esta discussão seja desde já proveitosa para qualquer leitor interessado em se aprofundar no paradigma de Orientação a Objetos – ou mesmo na linguagem Java, cujas *inner classes* já implementam uma forma restrita de closure.



O que são closures?

Segundo o FOLDOC (veja **Links**), closure é uma estrutura de dados que contém uma expressão e amarrações[1] de variáveis. Já a Wikipedia a define como uma função que é avaliada em um ambiente contendo uma ou mais variáveis amarradas. As definições mais gerais tratam qualquer elemento que possua um ambiente de variáveis fechado (daí o nome "closures") que seja um cidadão de primeira classe (veja o **quadro** "Cidadãos de primeira classe") como closures. Neste sentido, objetos Java e ponteiros para função em C poderiam ser chamados de closures.

Na prática, e em termos mais simples (talvez até simplista), closures são blocos de código (conjunto de expressões) que podem acessar variáveis do seu escopo e que podem ser criados dinamicamente, armazenados em variáveis e passados como parâmetros. A **Listagem 1** traz um exemplo bem simples de uso de closures na linguagem Groovy.

Nos exemplos de código, onde são mostradas closures em várias linguagens ou variantes de closures, destacaremos em negrito todas as closures; e em alguns casos outros elementos de linguagem relacionados ao suporte de closures.

Neste exemplo, uma closure que imprime **"Olá!"** na tela é definida e armazenada na variável **clos**. Quando esta variável é

1 Amarração (em inglês, "binding") é a associação entre entidades de programação, tal como, neste caso, entre o nome de uma variável e seu valor.

Resumo DevMan

De que se trata o artigo:

O que são closures, como funcionam as três principais propostas para adição de closures à linguagem Java e um resumo da discussão sobre o assunto. O artigo mostra como instalar os protótipos de cada proposta e escrever código com closures em Java.

Para que serve:

Para apresentar o leitor a um possível novo recurso da linguagem de programação Java, permitindo assim, que forme uma opinião mais bem fundamentada sobre o assunto.

Em que situação o tema é útil:

Caso o recurso de closures venha a ser incluído na linguagem Java ou o leitor necessite usá-lo em outra linguagem de programação que já o possua.

Closures em Java:

Closures são blocos de código (conjunto de expressões) que podem acessar variáveis do seu escopo e que podem ser criados dinamicamente, armazenados em variáveis e passados como parâmetros **(ver Listagens 1 a 3).** O Java implementa closures por meio de classes internas anônimas **(Listagem 4),** mas muitos consideram o recurso verboso e restrito. Portanto, há alguns anos surgiu a idéia de se adicionar closures à linguagem Java. Hoje, temos três propostas principais: BGGA **(Listagens 5 a 8),** FCM **(Listagens 9 e 10)** e CICE+ARM **(Listagens 11 e 12).** As propostas geraram muita discussão sobre o assunto, dividindo a comunidade em favoráveis e não-favoráveis à adição do recurso ao Java."

• A motivação para se adicionar closures é mais ligada à promoção da linguagem (marketing) do que realmente a benefícios para os programadores.

Para mais detalhes sobre a discussão, recomenda-se dois vídeos disponíveis no site *parleys.com*: *"Closures for Java"*, de Neal Gafter (um dos autores da proposta BGGA) e *"The Closures Controversy"*, de Joshua Bloch (um dos autores da proposta CICE+ARM).

Bloch cita nesta palestra um artigo de James Gosling, escrito em 1996, chamado *"The Feel of Java"*, em que James diz: *"Java é uma linguagem de colarinho azul. Não é material de uma tese de doutorado, mas uma linguagem para um trabalho."* Segundo Bloch, a adição de closures vai contra o "feel" da linguagem Java, tornando-a muito complexa para o trabalho do dia-a-dia. Em seu blog, Gosling, por sua vez, disse que sua fala foi mal interpretada, que Java não incluiu closures desde o início por falta de tempo e que closures já não são assunto de teses de doutorado. Confirmando isso, várias fontes[5] contam que Bill Joy, um dos designers originais do Java, era ferozmente favorável à inclusão de closures (e tipos genéricos) desde o Java 1.0; mas foi voto vencido apenas devido à urgência em finalizar logo a linguagem.

Enquanto isso, no site da proposta BGGA se pode ver um rascunho para uma JSR (Java Specification Request, ou Pedido de Especificação Java) para adição de closures em Java. Já a apóiam os autores de todas as três propostas (exceto Bloch e Gosling), empresas como BEA, Borland e JetBrains, além do SouJava. Alguns julgam que é cedo ainda para criar uma JSR, outros já dão como certa a inclusão de closures no Java 7. Aguardamos ansiosos os próximos capítulos desta história...

Para ajudar o leitor, proponho o seguinte exercício: analise o código da **Listagem 13** e tente adivinhar o que será impresso na tela. Em seguida, teste-o com o protótipo do BGGA. E prepare-se para provas de certificação bem mais difíceis.

5 Veja um artigo de Patrick Naughton em http://www.blinkenlights.com/classiccmp/javaorigin.html que conta a história da criação de Java. E para quem acha os tipos genéricos do Java 5 complicados ou limitados, isso decorre 90% da necessidade de compatibilidade com um enorme legado de APIs pré-Java 5.

Listagem 13. Quebra-cabeça Java: o que é impresso quando este programa é executado?

```
import java.util.*;

public class PuzzlerClosures {
  static void testeIb({ -> void } closure) {
    closure.invoke();
    int x = 20;
    closure.invoke();
    System.out.println(x);
  }
  static void teste1() {
    @Shared int x = 10;
    testeIb({ -> System.out.print((x++) + ", "); });
  }
  static void teste2() {
    List<{ -> int }> closures = new ArrayList<{ -> int }>();
    @Shared int i = 0;
    while (i++ < 10)
      closures.add({ -> i });
    int total = 0;
    for ({ -> int } c : closures)
      total += c.invoke();
    System.out.println(total);
  }
  public static void main(String[] args) {
    teste1();
    teste2();
  }
}
```

Conclusões

O debate continua e em breve talvez veremos um grupo de especialistas no JCP reunidos para elaborar uma proposta final de adição de closures em Java. Enquanto isso, nós desenvolvedores podemos instalar os protótipos, estudar as especificações, testá-las e emitir nossa opinião. Afinal, Java é construída pela sua comunidade! ●

Vítor E. Silva Souza
vitorsouza@gmail.com, labes.inf.ufes.br/vsouza/
é mestre em Informática com ênfase em Engenharia de Software pela UFES, com experiência em docência na área de Linguagens de Programação. Desenvolvedor Java desde 1999, especializou-se no desenvolvimento de aplicações Web, com as quais trabalha há 8 anos. É um dos fundadores do Grupo de Usuários Java do Estado do Espírito Santo (ESJUG).

Dê seu feedback sobre esta edição!

A Java Magazine tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/esmag/feedback

foldoc.org/foldoc.cgi?query=closure
Definição de closures segundo o Free Online Dictionary of Computing.

wikipedia.org/wiki/Closure_(computer_science)
Definição de closures segundo a Wikipedia.

gafter.blogspot.com/2007/01/definition-of-closures.html
Post no blog do Neil Gafter que traz a história das closures.

javac.info
Site da proposta BGGA.

openjdk.java.net/projects/closures
Subprojeto do OpenJDK que conduz a implementação da BGGA para o Java SE 7.

docs.google.com/View?docid=ddhp95vd_0f7mcns
Site da proposta FCM.

slm888.com/javac/
Site da proposta CICE+ARM.

parleys.com
Site onde se pode encontrar vídeos de apresentações sobre closures.

blogs.sun.com/jag/entry/closures
Post no blog do James Gosling sobre o debate das closures.

Closures

- Origens:
  - Década de 50, MIT: expressões lambda (funções anônimas);
  - Década de 70: Scheme fazia ligação estática, produzindo um fechamento (matemático).
- FOLDOC: “Uma estrutura de dados que contém uma expressão e um escopo de ligação de variáveis no qual a expressão será availada”;
- Wikipedia: “Uma função que é avaliada num escopo contendo uma ou mais variáveis ligadas”;

Closures

- Blocos de código que acessam variáveis em um escopo definido;
- São elementos de primeira classe:
  - Podem ser criados dinamicamente;
  - Podem ser armazenados em variáveis;
  - Podem ser passados como parâmetros;
  - Podem ser evocados dinamicamente.

Closures: exemplo

```
import static java.lang.System.out;
import java.util.function.Consumer;

public class Java8 {
  public static void main(String[] args) {
     String msg = "Hello, world!";
     Runnable clos1 = () -> out.println(msg);
     clos1.run();

     Consumer<String> clos2 =
          (String s) -> out.println(s);
     clos2.accept("Hello, world!");
  }
}

// Resultado:
// Hello, world!
// Hello, world!
```

Interfaces funcionais

- Considere as linhas:

```
Runnable clos1 = () -> out.println(msg);
Consumer<String> clos2 = (String s) -> out.println(s);
```

- As *closures* são "atalhos" para classes anônimas;
- Mas como o Java sabe que a 1ª é compatível com a interface Runnable e a segunda com Consumer<String>?
- O casamento é possível porque ambas são interfaces funcionais, i.e., interfaces com um único método:
  - Runnable tem void run();
  - Consumer<T> tem void accept(T t);
- Atribuir a *closure* a um Object causa erro de compilação.

Interfaces funcionais

- Muitas já definidas no pacote java.util.function. Exemplos:
  - Function<T, R>: recebe um T e retorna um R;
  - Supplier<T>: retorna um T;
  - Predicate<T>: retorna um booleano baseado no valor de um T;
  - Consumer<T>: consome um T;
  - BiFunction: como Function, mas com 2 parâmetros;
  - BiConsumer: como Consumer, mas com 2 parâmetros;
  - Etc.

Um caso mais real: Comparator (1)

```
class ComparadorTamanho implements Comparator<String> {
  public int compare(String s1, String s2) {
     if (s1.length() < s2.length()) return -1;
     if (s2.length() < s1.length()) return 1;
     return 0;
  }
}

public class Java8 {
  public static void main(String[] args) {
     String[] vet = new String[] {“ordenando", "de", "forma",
                                  "diferente"};
     Arrays.sort(vet);
     for (String s : vet) System.out.print(s + " ");
     System.out.println(); // de diferente forma ordenando

     Arrays.sort(vet, new ComparadorTamanho());
     for (String s : vet) System.out.print(s + " ");
     System.out.println(); // de forma diferente ordenando
  }
}
```

Os *imports* foram omitidos para classes bem conhecidas.

Um caso mais real: Comparator (2)

```
class ComparadorTamanho implements Comparator<String> {
  public int compare(String s1, String s2) {
     return Integer.compare(s1.length(), s2.length());
  }
}

public class Java8 {
  public static void main(String[] args) {
     String[] vet = new String[] {“ordenando", "de", "forma",
                                  "diferente"};
     Arrays.sort(vet);
     for (String s : vet) System.out.print(s + " ");
     System.out.println(); // de diferente forma ordenando

     Arrays.sort(vet, new ComparadorTamanho());
     for (String s : vet) System.out.print(s + " ");
     System.out.println(); // de forma diferente ordenando
  }
}
```

Um caso mais real: Comparator (3)

```
public class Java8 {
  public static void main(String[] args) {
     String[] vet = new String[] {"ordenando", "de", "forma",
                                  "diferente"};
     Arrays.sort(vet);
     for (String s : vet) System.out.print(s + " ");
     System.out.println(); // de diferente forma ordenando

     Arrays.sort(vet, new Comparator<String>() {
        public int compare(String s1, String s2) {
           return Integer.compare(s1.length(), s2.length());
        }
     });
     for (String s : vet) System.out.print(s + " ");
     System.out.println(); // de forma diferente ordenando
  }
}
```

Um caso mais real: Comparator (4)

```
public class Java8 {
  public static void main(String[] args) {
     String[] vet = new String[] {“ordenando", "de", "forma",
                                  "diferente"};
     Arrays.sort(vet);
     for (String s : vet) System.out.print(s + " ");
     System.out.println(); // de diferente forma ordenando

     Arrays.sort(vet,
        (s1, s2) -> Integer.compare(s1.length(), s2.length()));


     for (String s : vet) System.out.print(s + " ");
     System.out.println(); // de forma diferente ordenando
  }
}
```

Referências a métodos

- Closures são objetos de um método só;
- Às vezes o método que queremos colocar numa closure já existe:

```
Consumer<String> clos =
    (String s) -> System.out.println(s);
clos.accept("Hello, world!");
```

- Podemos ao invés disso simplesmente fazer referência ao método que já existe:

```
Consumer<String> clos = System.out::println;
clos.accept("Hello, world!");
```

Métodos default (1)

```
interface Mensagem { void dizer(); }

class OlaMundo implements Mensagem {
  public void dizer() {
     System.out.println("Olá, mundo!");
  }
}

class HelloWorld implements Mensagem {
  public void dizer() {
     System.out.println("Hello, world!");
  }
}

public class Java8 {
  public static void main(String[] args) {
     new OlaMundo().dizer();
     new HelloWorld().dizer();
  }
}
```

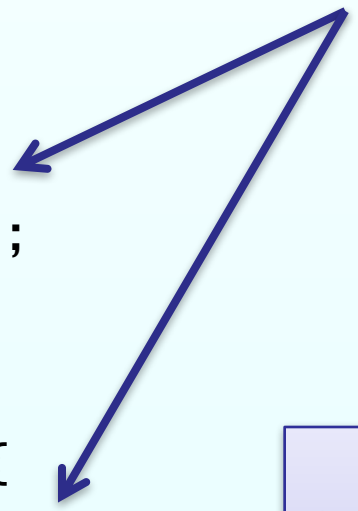
Métodos default (2)

```
interface Mensagem {
  void dizer();
  String getLang();
}

class OlaMundo implements Mensagem {
  public void dizer() {
     System.out.println("Olá, mundo!");
  }
}

class HelloWorld implements Mensagem {
  public void dizer() {
     System.out.println("Hello, world!");
  }
}

// ...
```



Não compilam mais!

Imagina se fazem isso com uma interface muito usada, tipo as da API do próprio Java!

Métodos default (3)

```
interface Mensagem {
  void dizer();
  default String getLang() { return null; }
}

class OlaMundo implements Mensagem {
  public void dizer() {
     System.out.println("Olá, mundo!");
  }
}

class HelloWorld implements Mensagem {
  public void dizer() {
     System.out.println("Hello, world!");
  }
}

// ...
```

Métodos default permitem implementar métodos em interfaces e herdá-los nas classes.

Exemplo: métodos default em java.util.List

- Com métodos default, novos e úteis métodos puderam ser adicionados à interface List;
- Outros: Collection.removeIf(), Comparator.revesed(), etc. (procure na API).

```
public class Java8 {
  public static void main(String[] args) {
    List<String> lista =
         Arrays.asList("Vitor", "Silva", "Souza");

    lista.sort((s1, s2) -> s1.compareTo(s2));

    lista.forEach(s -> System.out.print(s + " "));
    System.out.println();
  }
}
```

Métodos default e interfaces funcionais

- Interfaces funcionais são interfaces com um único método **não default**;
- Consumer<T>, que usamos antes, é um exemplo:
  - void accept(T t);
  - default Consumer<T> andThen(Consumer<? super T> after).

Retorna um Consumer composto que executa a operação do this seguida pela operação do after.

API de stream

- Coleções são muito úteis, porém possuem limitações. Por exemplo, não existem filtragem;
- Adicionar mais e mais funcionalidades nas coleções vai complicar a API. Foi criada então uma nova API: stream;
- Stream é uma interface do pacote java.util.stream;
- Representa uma sequência de objetos, como um iterador, porém suporta mudanças e paralelização.

API de stream: exemplo

```
public class Java8 {
  public static void main(String[] args) {
     List<String> lista =
          Arrays.asList("Vitor", "Silva", "Souza");

     lista.stream()
        .filter(s -> s.charAt(0) == 'S')
        .forEach(System.out::println);
  }
}

// Resultado:
// Silva
// Souza
```

API de stream: exemplo

```
public class Java8 {
  public static void main(String[] args) {
    List<String> lista = Arrays.asList("A", "AA",
"AAA");
    Stream<Integer> st =
lista.stream().map(String::length);
    st.forEach(System.out::println);
  }
}

// Resultado:
// 1
// 2
// 3
```

Optional

- Classe java.util.Optional;
  - Para evitar valores de retorno nulos (e consequentemente NullPointerExceptions);
  - Conceito já presente em Google Guava e Scala.
- NPEs são, muitas vezes, difíceis de depurar, quando o valor passa por vários métodos até ser usado;
- A ideia é pode diferenciar entre uma referência que, em um contexto específico, poderia potencialmente ser nula (e obrigar o programador a verificá-la);
  - Da mesma forma que exceções checadas nos obrigam a tratar erros.

Optional: exemplo (1)

```
class Pessoa {
  private String nome;
  public Pessoa(String n) { nome = n; }
  public String toString() { return nome; }
  public void ola() {
     System.out.println("Olá, " + nome + "!");
  }
}

class BancoDados {
  private Map<String, Pessoa> pessoas;
  public BancoDados(Collection<Pessoa> lista) {
     pessoas = new HashMap<>();
     lista.forEach(s -> pessoas.put(s.toString(), s));
  }
  public Pessoa procurar(String nome) {
     return pessoas.get(nome);
  }
}
```

Optional: exemplo (2)

```
public class Java8 {
  public static void main(String[] args) {
     List<Pessoa> lista = Arrays.asList(
        new Pessoa("Davi"), new Pessoa("Thiago"),
        new Pessoa("Daniel"), new Pessoa("Marcelo"));
     BancoDados bd = new BancoDados(lista);

     Pessoa p = bd.procurar("Daniel");
     p.ola();

     p = bd.procurar("David");
     p.ola();
  }
}

// Resultado:
// Olá, Daniel!
// Exception in thread "main" java.lang.NullPointerException
```

Optional: exemplo (3)

- Não está evidente pela assinatura que o método pode não encontrar uma pessoa e retornar nulo:

```
public Pessoa procurar(String nome) {
```

- Utilizamos, então, Optional para que isso fique

```
public Pessoa procurar(String nome) {
  Pessoa p = pessoas.get(nome);
  if (p == null) return Optional.empty();
  else return Optional.of(p);
}
```

```
public Pessoa procurar(String nome) {
  return Optonal.ofNullable(pessoas.get(nome));
}
```

Optional: exemplo (4)

- Obrigando o cliente a verificar:

```
Optional<Pessoa> p = bd.procurar("Daniel");
if (p.isPresent()) p.get().ola();

p = bd.procurar("David");
if (p.isPresent()) p.get().ola();
```

- E se você não gosta do if (e gosta de closures):

```
Optional<Pessoa> p = bd.procurar("Daniel");
p.ifPresent(Pessoa::ola);

p = bd.procurar("David");
p.ifPresent(Pessoa::ola);
```

Optional e stream

- Vários pontos da API agora utilizam Optional ou alguma classe derivada;
- Exemplo: qual a média de uma sequência vazia de valores? OptionalDouble evita erros:

```
import java.util.*;
import java.util.stream.*;

public class Java8 {
  public static void main(String[] args) {
     List<String> lista = Arrays.asList();
     IntStream st = lista.stream().mapToInt(String::length);

     OptionalDouble media = st.average();
     System.out.println(media.orElse(0));
  }
}
```

Nova API de datas/horas

- Em Java 1.0, existia só a classe Date;
- Como era complicada de manipular, Java 1.1 adicionou Calendar;
- Calendar não agradava muitos programadores. Surgiu a biblioteca JodaTime (joda.org/joda-time/);
- Com base em JodaTime, uma nova API de datas/horas vem sendo trabalhada para Java desde 2007;
- Finalmente esta API foi incorporada à API Java, no pacote java.time.

Nova API de datas/horas: exemplo (1)

- Datas para computadores: representação interna continua igual.

```
// 2014-06-26T22:16:30.175Z (formato ISO-8601)
Instant agora = Instant.now();
System.out.println(agora);

// Duração (ms): 5
Instant inicio = Instant.now();
for (int i = 0; i < Integer.MAX_VALUE; i++);
Instant fim = Instant.now();
Duration duracao = Duration.between(inicio, fim);
long duracaoEmMilissegundos = duracao.toMillis();
System.out.println("Duração (ms): " + duracaoEmMilissegundos);

// Porque não era necessário mudar a representação:
// Sun Aug 17 04:12:55 BRT 292278994
System.out.println(new Date(Long.MAX_VALUE));
```

Nova API de datas/horas: exemplo (2)

- Datas para seres humanos: substitui a classe Calendar;
- Dá suporte a diferentes calendários.

```
LocalDate hoje = LocalDate.now();
System.out.println(hoje);   // 2014-06-26 (formato ISO-8601)

LocalDate lancamentoJava8 = LocalDate.of(2014, 3, 18);
// Ou: lancamentoJava8 = LocalDate.of(2014, Month.MARCH, 18);
System.out.println(lancamentoJava8);  // 2014-03-18

// 3 meses e 8 dias pra eu falar de Java 8
Period periodo = Period.between(lancamentoJava8, hoje);
System.out.printf("%s meses e %s dias pra eu falar de Java
    8%n", periodo.getMonths(), periodo.getDays());

LocalTime horarioDeEntrada = LocalTime.of(9, 0);
System.out.println(horarioDeEntrada); // 09:00

LocalDateTime aberturaDaCopa = LocalDateTime.of(2014,
Month.JUNE, 12, 15, 0);
System.out.println(aberturaDaCopa);   // 2014-06-12T15:00
```

http://nemo.inf.ufes.br/