Linguagem Java:

Um Passeio pela Linguagem

Rômulo Silva de Oliveira
Instituto de Informática - UFRGS

romulo@inf.ufrgs.br
http://www.inf.ufrgs.br/~romulo

Um Passeio pela Linguagem Java

- Programas são construídos a partir de classes
- Classes são moldes/formas para criar objetos
- Objetos são instâncias de classes
- Classes possuem dois tipos de membros:
  - Campos (atributos, dados, estado)
  - Métodos (procedimentos, instruções)

Aplicação AloMundo

```
class AloMundo {
   public static void main( String[] args){
        System.out.println("Alô Mundo");
        }
   }
```

- Programa com uma única classe **AloMundo**, que possui zero campos e um método (**main**)
- Método main possui um parâmetro: **args** do tipo array de **String**
- **void** significa que **main** não retorna valor
- **public static void main** - método chamado quando a aplicação iniciar
- **System.out.println** escreve uma linha na tela
- Nomes de classes iniciam com Maiúsculas
- Nomes de campos e métodos iniciam com minúsculas
  - Podem ter Maiúsculas no meio para facilitar a leitura

Exemplo - Série de Fibonacci

```
class Fibonacci {
   /** Mostra serie Fibonacci ate 50 */
   static final int MAXIMO = 50;
   public static void main( String[] args){
      int ti=1;
      int tj=1;

      System.out.println(ti);
      while(tj<MAXIMO){
        System.out.println(tj);
        tj=ti+tj; //novo tj
        ti=tj-ti; //ti recebe antigo tj
      }
   }
}
```


Exemplo - Série de Fibonacci

- **ti** e **tj** são variáveis locais do método **main**
- Todas as variáveis são tipadas
- **while** funciona como esperado
- Variáveis podem ser inicializadas na declaração
- Compilador detecta uso de variável não inicializada
- **println** aceita argumentos de diferentes tipos
- **static final** - Cria uma variável que não pode ser alterada e possui valor único para todas as instâncias da classe: uma constante
- Comentários com // até o final da linha
- /* como na linguagem C */
- /** preservado por geradores automaticos de
- documentação */

Exemplo - Série de Fibonacci

- Tipos de dados primitivos de Java:

- **boolean** - true ou false
- **char** - caracter Unicode 1.1 de 16 bits
- **byte** - inteiro de 8 bits com sinal
- **short** - inteiro de 16 bits com sinal
- **int** - inteiro de 32 bits com sinal
- **long** - inteiro de 64 bits com sinal
- **float** - ponto flutuante de 32 bits (IEEE)
- **double** - ponto flutuante de 64 bits (IEEE)

Classes e Objetos

```
class Ponto {
  public double x, y;
}
```

- **Ponto** é uma classe sem métodos
- **public** indica que qualquer código
  com acesso a um objeto da classe Ponto
  pode ler ou modificar o campo

```
Ponto esqInf = new Ponto();
Ponto dirSup = new Ponto();
esqInf.x = 0.0;          esqInf.y = 0.0;
dirSup.x = 1280.0;       dirSup.y = 1024.0;
```


Classes e Objetos

- Objetos são criados com **new** (instanciação)
- Objetos são sempre acessados através de referências a objetos
  - **esqInf** é uma referência
- Campos em objetos são variáveis de instância
  - Existe uma versão para cada instância de objeto

Campo Estático

- Campo estático ou variável de classe ou campo de classe
  - Campo válido para a classe como um todo
    e não para cada instância separadamente
- Exemplo:
  - Classe walkman
  - Cada instância contém os dados de um aparelho em particular,
    inclusive número de série único
  - Qual o próximo número de série a ser usado ?
    guarda como uma variável de classe

```
class Walkman {
  public static int numeroDeSerie = 0;
  ...
}
```


Método Estático

- Método estático ou método de classe:
  - Método válido para a classe como um todo
    e não para cada instância separadamente
- Métodos normais:
  - São executados dentro do contexto de uma instância específica,
    dependem da criação de instâncias
- Métodos estáticos:
  - São executados no contexto da classe como um todo,
    NÃO dependem da criação de instâncias

- Método estático SOMENTE pode acessar
  membros (campos e métodos) estáticos

- Exemplo: **main**, **Math.sqrt**

Arrays

```
class Baralho {
  final int TAM_BARA = 52;
  carta[] cartas = new Carta[TAM_BARA];

  public void imp() {
   for(int i=0; i<cartas.length; ++i)
       System.out.println(cartas[i]);
  }
}
```

- Todo array possui um campo **length** com o número de elementos
- Os limites são **0** e **length-1**
- Exceção é lançada se índice está fora dos limites
- Variável declarada na inicialização do **for**
  - Válida somente dentro do **for**

Strings

```
class DemoStrings {
 static public void main( String args[]){
   String meuNome = "Pedro";
   meuNome = meuNome + " Paulo";
   System.out.println("Nome="+meuNome);
   }
}
```

- **String** é objeto e não tipo primitivo
- Não precisa especificar tamanho ao criar
- Operador + usado para concatenar strings
- Objeto **String** possui método **length()**
  que retorna o número de caracteres

Strings

- Objetos **String** são somente para leitura:

```
String str;
str = "azul";
str = "vermelho";
```

- **str** contém referência para objeto
  e não o objeto propriamente dito

- Método **equals** usado para comparação

```
if( umStr.equals( outroStr ) )
    ...
```


Estendendo uma Classe

- Característica importante da orientação a objetos:
  Estender classe existente, criando subclasses
- A subclasse herda todos os métodos e campos da classe original
  - Chamada superclasse
- A subclasse pode sobrepor métodos
  - Substitui o método da superclasse por um novo

- Uma classe (ex: Carro) pode ser subclasse de
  apenas uma superclasse (ex: Veiculo)
- Uma classe (ex: Veiculo) pode ser superclasse de
  várias subclasses (ex: Carro, Onibus, Caminhao)
- Subclasse pode ser usada em qualquer lugar
  onde a superclasse era esperada

Estendendo uma Classe - Exemplo

```
class Ponto {
   public double x, y;
   public void limpar(){
       x = 0.0;      y = 0.0;
   }
}
class Pixel extends Ponto {
   public Color cor;
   public void setCor( Color x){
       this.cor = x;
   }
   public void limpar() {
       super.limpar();      cor = null;
   }
}
```


Estendendo uma Classe - Exemplo

- **Pixel** é uma subclasse de **Ponto**
- **Ponto** é a superclasse de **Pixel**
- Classe **Pixel** sobrepôs o método **limpar()** da classe **Ponto**
- **super** faz referência a coisas da superclasse
- **this** faz referência a coisas do próprio objeto

Estendendo uma Classe - Polimorfismo

- Um objeto **Pixel** pode ser usado no lugar de um objeto **Ponto**
  - Polimorfismo

```
Ponto po;          Pixel pi = new Pixel();
po = pi;
po.limpar();
```

- **pi** faz referência a objeto do tipo **Pixel**,
  que pode ser usado como se fosse **Ponto**

- Vale o método do tipo real do objeto
  e não do tipo da referência usada:
  - Vai executar o **limpar()** de **Pixel**

Estendendo uma Classe - Polimorfismo

- Objeto **Ponto** NÃO pode ser usado no lugar de um objeto **Pixel**

```
Pixel pi;          Ponto po = new Ponto();
pi = po;  //ERRO!
pi.setCor( null);
```

- Toda classe que
  Não estende explicitamente outra classe,
  Estende implicitamente a classe **Object**

```
Object oref;
oref = "texto";
oref = new Pixel();
```

Pacotes

- Utilizado para resolver os conflitos de nomes
- Nomes hierárquicos, componentes separados por pontos (.)
- Para usar uma parte de um pacote,
  é possível usar o nome completo

```
class Data1 {
  public static void main( String[] args) {
      java.util.Date agora =
                        new java.util.Date();
      System.out.println(agora);
  }
}
```


Pacotes

- Também é possível importar o pacote,
  tornando visíveis seus componentes

```
import java.util.Date
class Data1 {
  public static void main( String[] args) {
      Date agora = new Date();
      System.out.println(agora);
  }
}
```

- É possível importar todo o conteúdo de um pacote:
  import java.util.*

Pacotes - Nomes

- Dois pacotes ainda assim podem ter o mesmo nome
- Solução: Utilizar como prefixo do nome do pacote
  o nome de domínio na Internet ao contrário

- Exemplo: Empresa xsoftware.com escreve pacote roleta
  - **COM.xsoftware.roleta**
- Informática da UFRGS escreve pacote roleta
  - **BR.ufrgs.inf.roleta**

- Pacote é criado com a declaração no início do arquivo fonte:
  package BR.ufrgs.inf.roleta;

- Se package não é usado, vira um pacote sem nome
- Biblioteca de Java formada por dezenas de pacotes

Linguagem Java:

Tipos, Operadores e Expressões

Rômulo Silva de Oliveira
Instituto de Informática - UFRGS

romulo@inf.ufrgs.br
http://www.inf.ufrgs.br/~romulo

Tipos, Operadores e Expressões

- Programas Java são escritos em Unicode
  - Conjunto de caracteres com 16 bits
  - Primeiros 256 caracteres iguais a Latin-1
  - Primeiros 128 caracteres semelhantes a ASCII
- Editores de texto não trabalham com Unicode
  - **\udddd**, onde dddd são 4 dígitos hexadecimais
- Caracteres especiais:
  \n \t \b \r \f \\ \' \"
  \ddd
- Inteiros podem ter vários formatos:
  29 035 0x1D 0X1d
  - (sufixo **L** ou **l** indicam tipo long)

Tipos, Operadores e Expressões

- Ponto flutuante também:
  18. 1.8e1 .18E2
  - Sufixo **F** ou **f** indicam tipo float

- Valor inicial default para campos de classes:
  - referência a objeto: **null**
  - boolean: **false**
  - char: **'\u0000'**
  - inteiro: **0**
  - ponto flutuante: **+0.0**
- Variáveis locais NÃO possuem inicialização default

Procura por um Identificador

- O significado de um nome é procurado na ordem:

1. Variáveis locais declaradas em Bloco, For,
   Manipulador de exceções
2. Parâmetros do método ou construtor
3. Campos e métodos da classe, inclusive herdados
4. Tipos importados explicitamente nomeados
5. Outros tipos declarados no mesmo pacote
6. Tipos importados implicitamente nomeados
7. Pacotes disponíveis no computador

Arrays

- Criação de arrays:

```
Atr[] atrs = new Atr[12];          // ou
Atr[] atrs;     atrs = new Atr[12];
```

- Posições do array **atrs** estão inicialmente vazias

```
for(int i=0;i<atrs.length; i++)
  atrs[i] = new Atr(algumParametro);
```

- Arrays também podem ser inicializados

```
String[] cores = {"azul","branco","preto"};    // ou
String[] cores = new String[3];
...
cores[0]="azul";   cores[1]="branco";
cores[2]="preto";
```


Array de Arrays

```
float[] [] mat = new float[4][4];
...
for(int y=0; y<mat.length;y++) {
  for(int x=0;x<mat[y].length; ++x)
      System.out.print(mat[x][y]+" ");
  System.out.println();
}
```

- Primeira dimensão deve ser especificada
  - Quando o array é criado
- Outras dimensões podem ficar sem especificação
  - Para serem definidas mais tarde

Array de Arrays

```
float [] [] mat = new float[4][];
...
for( int y=0;y<mat.length;y++)
  mat[y] = new float[y+10];
```

- Cada array aninhado pode ter um tamanho diferente: 10, 11, 12 e 13

- Inicialização também é possível

```
double[][] mat = {
  { 0.0, 0.1, 0.2, 0.3 } ,
  { 1.0, 1.1, 1.2, 1.3 } ,
  { 2.0, 2.1, 2.2, 2.3 } ,
  { 3.0, 3.1, 3.2, 3.3 }
  };
```


Tipos de Expressões

- Operações aritméticas são efetuadas com o tipo do maior operando:
  - **double**
  - **float**
  - **long**
  - **int**
  - (**char**, **short** e **byte** são convertidos para **int**)

- Quando um dos operandos for do tipo **String**,
  na operação "+" ou "+="
  o outro operando é convertido para **String**

Conversões de Tipos

- Conversão Implícita:
  - Quando um valor de tipo "menor" é atribuido ou
    operado com uma variável ou um valor de tipo "maior"

- Conversão Explícita: **Cast**

```
double d = 7.99;
long l = (long) d;
```

- Ponto flutuante para inteiro TRUNCA
- Classe **Math** permite arredondamento

Conversões de Referências a Objetos

```
class Cafe {            \\ Superclasse
...
class CafeExpresso extends Cafe {   \\ Subclasse
...
class CafeComLeite extends Cafe {   \\ Subclasse
...
```

- Referência para superclasse recebe subclasse

```
Cafe c;
CafeExpresso exp = new CafeExpresso();
c = exp;          \\ OK, Cast Up
```


Conversões de Referências a Objetos

- Referência para subclasse recebe superclasse

```
CafeComLeite cl;
Cafe c = new Cafe();
cl = c;           \\ Cast Down invalido
```

- Referência para subclasse recebe superclasse

```
CafeComLeite cl2 = new CafeComLeite();
CafeComLeite cl;
Cafe c;
c = cl2;                \\ OK, Cast Up
cl2 = (CafeComLeite) c; \\ Cast Down ok
if(c instanceof CafeComLeite)
  ...                   \\ Será executado
```


Strings

- Construtores:

```
String s;
s = "texto";
s = String();           \\ retorna ""
```

- Métodos:

```
s.length()                  s.charAt(int pos)

s.indexOf(char ch)
s.indexOf(char ch, int inic)
s.indexOf(String str)
s.indexOf(String str, int inic)
```


Strings

```
s.lastIndexOf(char ch)
s.lastIndexOf(char ch, int fim)
s.lastIndexOf(String str)
s.lastIndexOf(String str, int fim)

s.equals(String str)
s.equalsIgnoreCase(String str)
s.compareTo(String str)

s.replace(char ant, char novo)
s.toLowerCase()
s.toUpperCase()
s.trim()
```


Conversões para String

```
String.valueOf(boolean)

String.valueOf(int)

String.valueOf(long)

String.valueOf(float)

String.valueOf(double)

String.copyValueOf(char[],inic,comp)
```


Conversões de Strings

```
new Boolean(String).booleanValue()

Integer.parseInt(String, int base)

Long.parseLong(String, int base)

new Float(String).floatValue()

new Double(String).doubleValue()

s.toCharArray()
```

Ordem de Avaliação dos Operadores

• Operadores pós-fixados	`[] . (params) exp++ exp--`
• Operadores Unários	`+exp -exp ++exp --exp ~ !`
• Criação ou Cast	`new (tipo) exp`
• Multiplicativos	`* / %`
• Aditivos	`+ -`
• Deslocamentos	`<< >> >>>`
• Relacionais	`< > >= <= instanceof`
• Igualdade	`== !=`
• E bit a bit	`&`
• Ou exclusivo bit a bit	`^`
• Ou bit a bit	`|`

Ordem de Avaliação dos Operadores

• E lógico	`&&`
• Ou lógico	`||`
• Condicional	`?:`
• Atribuição	`= += -= *= /= %= >>= <<= >>>= &= ^= |=`

- Avaliação da esquerda para a direita
- Todos operandos avaliados antes da operação, exceção para &&, || e ?:

Linguagem Java:

Controle de Fluxo

Rômulo Silva de Oliveira
Instituto de Informática - UFRGS

romulo@inf.ufrgs.br
http://www.inf.ufrgs.br/~romulo

Controle de Fluxo

- Instruções:
  - Instruções de Declaração
  - Instruções de Expressão
    - Atribuição
    - Incremento (++) e decremento (--)
    - Chamada de método
    - Criação de objeto
- Variáveis podem ser declaradas e inicializadas em qualquer ponto de um bloco

Comando IF

```
if ( expressão booleana )
  instrução;
else
  instrução;
```

- Cuidado com IF embutido

```
if ( valores.length > 1 ) {
    if ( x > y )
          soma = soma + x;
    }
else
    soma = 0;
```


Comando Switch

```
int x;
...
switch( x ) {
  case 0:   valor = media();
            break;
  case 1:   valor = media() * 2;
            break;
  default:  valor = 0;
            break;
  }
```

- Sem o BREAK a execução "escorrega" para baixo

Comando Switch

```
public int valHexa( char ch) {
  switch(ch) {
   case '0':        case '1':          case '2':
   case '3':        case '4':          case '5':
   case '6':        case '7':          case '8':
   case '9':  return( ch - '0');
   case 'a':        case 'b':          case 'c':
   case 'd':        case 'e':          case 'f':
            return( ch - 'a') + 10;
   case 'A':        case 'B':          case 'C':
   case 'D':        case 'E':          case 'F':
            return( ch - 'A') + 10;
   default:   return 0;
       }
}
```


Comando WHILE e DO-WHILE

- Comando WHILE

```
x = 0;
while( x < y ) {
  isto();
  aquilo();
  }
```

- Comando DO-WHILE

```
x = 0;
do {
      isto();
      aquilo();
      }while( x < y);
```


Comando FOR

```
for( inicialização, teste, ajuste)
            instrução;
```

- Forma compacta equivalente a:

```
{
  inicialização;
  while( teste ) {
      instrução;
      ajuste;
      }
  }
```


Comando FOR

- Vale qualquer coisa

```
for( i=0,j=arr.length-1; j>=0; i++,j--)
      metodoTal();

for( ; ; ){ // Loop infinito
  ...
  }
```

- É possível colocar vários comandos na inicialização e no ajuste

Comando CONTINUE

- Exemplo: Calcula menor potência de 10 que seja maior que o valor fornecido (p/escala em gráfico)

```
public int potDez( int valor) {
  int exp, v;

  for(exp=0,v=valor-1;v>0;exp++,v/=10)
      continue;
  return exp;
  }
```


Comando CONTINUE

- **CONTINUE** vale para:
  FOR, **WHILE** e **DO-WHILE**

```
while( !stream.eof() ) {
  token = stream.next();
  if( token.equals("skip") )
      continue;
  processa( token);
  ...
  }
```

Comando BREAK

- Utilizado para interromper: SWITCH, FOR, WHILE, DO-WHILE
- Interrompe o comando mais interno

```
for(i=0; i<objetos.length; i++)
    if(objetos[i] == null )
          break;
```

- Pode ser utilizado com rótulos

```
busca:
  for(y=0;y<matriz.length;++y)
      for(x=0;x<matriz[y].length;++x)
            if(matriz[y][x]==flag){
                  achou = true;
                  break busca;
                  }
```


Comando RETURN

- Deve combinar com o tipo retornado pelo método

```
private double naoNegativo(double val){
  if( val<0)
      return 0;
  else
      return val;
  }
```

- RETURN em construtor ou inicializador estático
  nunca retorna valor

Exceções

- Exceções em java são objetos que estendem a classe **Throwable**
- É lançado quando uma condição de erro é encontrada
- Pode ser capturada ou propagada
- Novos tipos de exceção estendem a classe **Exception**
- Objetos da classe **Exception** são exceções testadas
  - Para poder lançar esta exceção o método precisa declarar antes que poderá lançar esta exceção
- Objetos das classes **RunTimeException** e **Error** são exceções não testadas
  - Método não precisa declarar, teste automático

Comandos TRY, CATCH e FINALLY

```
try
    comandos
catch(tipo-de-exceção identificador)
    comandos
catch(tipo-de-exceção identificador)
    comandos
...
finally
    comandos
```


Comandos TRY, CATCH e FINALLY

- **finally** sempre executa, com ou sem exceção
- **catch** examinado um após o outro,
  até encontrar um tipo-de-exceção apropriado
- Somente uma cláusula **catch** é executada
- **tipo-de-exceção** deve ser
  - o tipo lançado ou
  - uma superclasse do tipo lançado
- Na lista de clausulas catch, subclasses devem vir antes
- É possível **try-catch** sem finally
  e também **try-finally** sem catch

Linguagem Java:

Classes e Objetos

Rômulo Silva de Oliveira
Instituto de Informática - UFRGS

romulo@inf.ufrgs.br
http://www.inf.ufrgs.br/~romulo

Classes e Objetos

- Objeto: O que faz = assinatura + semântica
  Como faz = sua classe

```
class CorpoCeleste {
   public long numId;
   public String nome;
   public CorpoCeleste orbita;
   public static long proxId;
     }
CorpoCeleste wolf359;
```

- **wolf359** é uma variável capaz de conter uma referência para objeto da classe **CorpoCeleste**
- **wolf359** é inicializado com **null**
  - **wolf359.numId = 0** é um ERRO!
  - Nenhum objeto **CorpoCeleste** foi criado

Controle de Acesso aos Campos e Métodos

- Todos os campos e métodos de uma classe estão sempre disponíveis ao código da própria classe

- Para controlar o acesso a partir de outras classes e para controlar a herança pelas subclasses
  - Existem 4 modificadores de controle de acesso:

- **public**
- **package** (default)
- **protected**
- **private**

Modificadores para Controle de Acesso

- **public**
  - É acessível em qualquer lugar onde a classe é acessível
  - São herdados pelas subclasses

- **protected**
  - É acessível e herdado por subclasses
  - É acessível para código no mesmo pacote

- **package** (sem modificador)
  - É acessível por código no mesmo pacote
  - É herdado apenas por subclasses no mesmo pacote

- **private**
  - É acessível apenas na própria classe

Construtores

- Um objeto recém criado recebe um estado inicial:
  - Inicialização default
  - Atribuição na declaração
  - Código de criação (construtores)

- Construtores não possuem tipo de retorno
- Código executado após as variáveis de instância terem sido inicializadas
- Construtor é chamado através do **new**

Construtores

```
class CorpoCeleste {
   public long numId;
   public String nome = "Sem nome";
   public CorpoCeleste orbita = null;

   private static long proxId = 0;

   CorpoCeleste() {
       numId = proxId++;
   }
}
```


Usando Construtores

- Caso comum: Criar o objeto e inicializar seus campos

```
CorpoCeleste sol = new CorpoCeleste();
sol.nome = "Sol";
...
CorpoCeleste terra = new CorpoCeleste();
terra.nome = "Terra";
terra.orbita = sol;
```

- Possível criar construtor para este caso
- Linguagem fornece um construtor padrão sem argumentos
- Se nenhum construtor é definido pelo programador o construtor padrão é usado

Definindo Construtores

```
class CorpoCeleste {
   public long numId;
   public String nome = "Sem nome";
   public CorpoCeleste orbita = null;
   private static long proxId = 0;

   CorpoCeleste() {
       numId = proxId++;
   }
   CorpoCeleste( String nomeCorpo,
              CorpoCeleste orbitaAoRedor) {
       this();
       nome = nomeCorpo;    orbita = orbitaAoRedor;
   }
}
```


Definindo Construtores

```
CorpoCeleste sol = new CorpoCeleste("Sol",null);
CorpoCeleste terra =
                  new CorpoCeleste("Terra",sol);
```

- Um construtor pode chamar outro construtor
  a partir da mesma classe
  usando **this()**
  como sua primeira instrução executável
- Pode chamar construtor com ou sem parâmetros

Métodos

- Chamadas como operações em objetos por meio de referências e usando o operador ponto: **referencia.metodo(parametros)**
- Java NÃO permite métodos com número variável de parâmetros
  - Cada parâmetro possui um tipo específico
- Método **tostring()** é especial:
  - Chamado para obter um **String** quando o objeto é utilizado em concatenação: **System.out.println("Corpo " + sol);**

```
public String toString() {
   String desc = numId + " (" + nome + ")";
   if( orbita != null )
       desc += " orbita " + orbita.toString();
   return desc;
}
```


Parâmetros e Valor de Retorno

- Todos os parâmetros são passados por valor
  - Método utiliza uma cópia local do valor

```
class PassagemPorValor {
   public static void main( String[] args) {
      double um = 1.0;
      System.out.println("antes:"+um);
      metade(um);
      System.out.println("depois:"+um);
   }

   public static void metade( double arg) {
      arg /= 2.0;
      System.out.println("metade:"+arg);
   }
}
```


Parâmetros e Valor de Retorno

- Referência a objeto:
  - A referência é passada por valor, o objeto mesmo é acessado
- Métodos retornam diretamente apenas um ou nenhum valor
- Para retornar vários valores:
  Retornar um objeto

this

- **this** pode ser usado para fazer referência a própria instância onde ele aparece:
  - **servico.acrescentar(this);**
- Pode estar explícito ou implícito no código

```
class Nome {
   public String str;
   Nome() {
       str = "sem nome";
             ou
       this.str = "sem nome";
   }
}
```

Sobrecarregando Métodos - Overloading

- Cada método possui uma assinatura
  assinatura = nome + número e tipo dos parâmetros
- Dois métodos podem ter o mesmo nome,
  se tiverem assinaturas diferentes
- Método chamado é identificado pela assinatura
  e não somente pelo nome

```
public CorpoCeleste orbitaAoRedor(){
      return orbita;
  }
public void orbitaAoRedor( CorpoCeleste aoRedor){
      orbita = aoRedor;
  }
```


Membro Estático

- Membro estático: Apenas 1 por classe
- Como inicializar campos estáticos ?
  - Bloco de inicialização estático

```
class Primos {
  protected static int[] numPrimos= new int[4];

  static {
      numPrimos[0] = 2;
      for(int i=1;i<numPrimos.length; i++)
            numPrimos[i]=proxPrimo();
  }
}
```


Membro Estático

- Inicialização estática segue o fonte
- Inicializadores de campos estáticos não podem
  chamar métodos que podem lançar exceções testadas
- Blocos de inicialização estática podem,
  desde que capturem a exceção
- Métodos estáticos são associados com a classe
  e não com as instâncias:
  - Acessa apenas membros estáticos
  - Não existe **this**
  - Chamado com o nome da classe:
    NomeClasse.metodoEstatico()

Método main( )

- Uma classe deve ser a classe principal da aplicação
- Esta classe é localizada e seu método **main()** é executado
- **main** deve
  - Ser **public static void**
  - Receber **String[]** como parâmetro

```
class Eco {
  public static void main( String[] args) {
    for(i=1;i<args.length;i++)
      System.out.println(args[i]+" ");
    System.out.println();
  }
}
```


Método main( )

- **args** são os argumentos do programa
  - Digitados na linha de comando ou
  - Definidos na interface gráfica de usuário
- Nome da classe não é incluido
- Todas as classes podem ter **main**
- Apenas o **main** da classe principal é chamado

Coleta de Lixo

- Quando um objeto não é mais referenciado
  o espaço que ele ocupa é liberado
- Automático, NÃO existe **delete**, **free**, **dispose**, etc
- Método **finalize**:
  executado antes do espaço ser recuperado
  ou
  quando a máquina virtual encerra a execução
- Utilizado para liberar recursos não-java
  quando o objeto é eliminado
- Exemplo: fechar arquivos, periféricos

Linguagem Java:

Estendendo Classes

Rômulo Silva de Oliveira
Instituto de Informática - UFRGS

romulo@inf.ufrgs.br
http://www.inf.ufrgs.br/~romulo

Estendendo Classes

- Uma classe que não estende nenhuma
  estende implicitamente a classe **Object**
- Variáveis tipo **Object** podem receber referências
  para objetos de qualquer tipo

```
class Atr {
  private String nome;
  private Object valor = null;

  public Atr(String nomeDe, Object valorDe) {
      nome = nomeDe;
      valor = valorDe;
  }
```


Estendendo Classes

```
  public String nomeDe() {
      return nome;
  }
  public Object valorDe() {
      return valor;
  }
  public Object valorDe( Object novoValor) {
      Object valorAnt = valor;
      valor = novoValor;
      return valorAnt;
  }
}
```


Estendendo Classes

- Uma estensão da classe **Atr**

```
class CorAtr extends Atr {
  private CorTela minhaCor;

  public CorAtr(String nome, Object valor) {
      super( nome, valor);
      minhaCor = new CorTela( valor);
  }
  public CorAtr(String nome) {
      this( nome, "transparente");
  }
  public CorTela cor() {
      return minhaCor;
  }
}
```


Construtores

- Construtor da subclasse deve chamar
  um dos construtores da superclasse
- **super( )** chama um construtor da superclasse

- Quando **super( )** não é a primeira instrução executável do construtor:
  - Construtor sem argumentos da superclasse
    é automaticamente chamado, se existir
  - Se não existir, erro

- **this( )** pode ser usado para chamar construtor do próprio objeto

- Construtores são tão públicos quanto as classes

Sobreposição - Override

- Substitui a implementação da superclasse de um método
  por um de seus próprios métodos
- Assinaturas devem ser idênticas
- Deve retornar o mesmo tipo de dado
- Pode reduzir as exceções possíveis,
  mas não especificar novas exceções
- O acesso pode ser ampliado mas não reduzido
- Exemplo:
  protected pode passar para **public**
  mas NÃO para **private**

Sobreposição - Override

- Campos NÃO podem ser sobrepostos, mas podem ser ocultos
- Se a subclasse declara um campo
  com o mesmo nome do campo da superclasse:
  - Campo da superclasse continua a existir
  - Nome usado na subclasse refere-se ao campo da subclasse
  - **super.nome** acessa campo da superclasse
- Pouco usado
- Campos devem ser privados sempre que possível
- Permitido para facilitar
  - a manutenção de superclasses
    sem invalidar as suas subclasses existentes

Tipo Efetivo e Tipo Declarado

- Chamada de método: Vale o tipo efetivo da referência
- Acesso a um campo: Vale o tipo declarado da referência

```
class SuperMostra {
  public String str = "SuperStr";

  public void mostra() {
    System.out.println("Super.mostra:" + str);
  }
}
```


Tipo Efetivo e Tipo Declarado

```
class EstendMostra extends SuperMostra{
  public String str = "EstendeStr";

  public void mostra() {
    System.out.println( "Estende.mostra:" + str);
  }

  public static void main(String[] args){
      EstendeMostra est = new stendeMostra();
      SuperMostra sup = est;
      sup.mostra();       est.mostra();

      System.out.println(sup.str + ";" + est.str);
  }
}
```


final

- Quando um método é **final**
  - Nenhuma classe estendida pode sobrepor o método
  - Versão final do método
- Classes inteiras podem ser definidas como final:
  final class NaoEstensivel { . . .
- Impede alterações que poderiam causar problemas
- Permite otimização

Classe Object

- Todas as classes estendem **Object**, herdam os métodos de **Object**
- **public boolean equals(Object obj)**
- **public int hashcode( )**
- **public Object clone( ) throws CloneNoSupportedException**
- **public final Class getClass( )**
- **protected void finalize( ) throws Throwable**
- **equals( )** é **true** apenas para o mesmo objeto
- **hashcode( )** fornece valores distintos para objetos diferentes,
  independente do conteúdo

Métodos e Classes Abstratas

- São classes que definem apenas parte da implementação
- Suas subclasses devem fornecer a implementação específica
  para os métodos que faltam
- Uma classe com métodos abstratos
  deve ser declarada como **abstract**
- É possível sobrepor métodos da superclasse
  para declara-los **abstract**
- NÃO é possível criar instâncias de uma classe abstrata

Classe Abstrata - Exemplo

```
abstract class Benchmark {
  abstract void benchmark();

  public long repetir(int cont) {
      long inic = System.currentTimeMillis();
      for(int i=0; i<cont; i++)
            benchmark();
      return System.currentTimeMillis() - inic;
  }
}
```


Classe Abstrata - Exemplo

```
class BenchmarkMethod extends Benchmark{
  void benchmark() {
      ...
  }

  public static void main( String[] args) {
      int cont = Integer.parseInt(args[0]);
      long tempo =
            new BenchmarkMethod().repetir(cont);
      System.out.println( ... );
  }
}
```


Interfaces

- Java possui herança simples de implementação
  - Subclasse pode estender uma única classe
- Interface:
  Uma classe completamente abstrata,
  sem implementação
- Java possui herança múltipla de interfaces
- Interface contém apenas
  - métodos **public** sem implementação
  - e constantes

Interfaces

```
interface Atributos {
  void adicionar(Atr novoAtr);
  Atr localizar(String nomeAtr);
  Atr remover(String nomeAtr);
  java.util.Enumeration atrs();
}
...
class CorpoAtributos
  extends CorpoCeleste
  implements Atributos {
  ...
```


Estendendo Interfaces

- Interfaces podem ser estendidas com a palavra chave extends
- Interfaces podem estender mais do que uma interface
  - Herança múltipla

```
interface Oferta
      extends ProdAlimenticios, ProdBazar {
  double precoOferta();
}
```