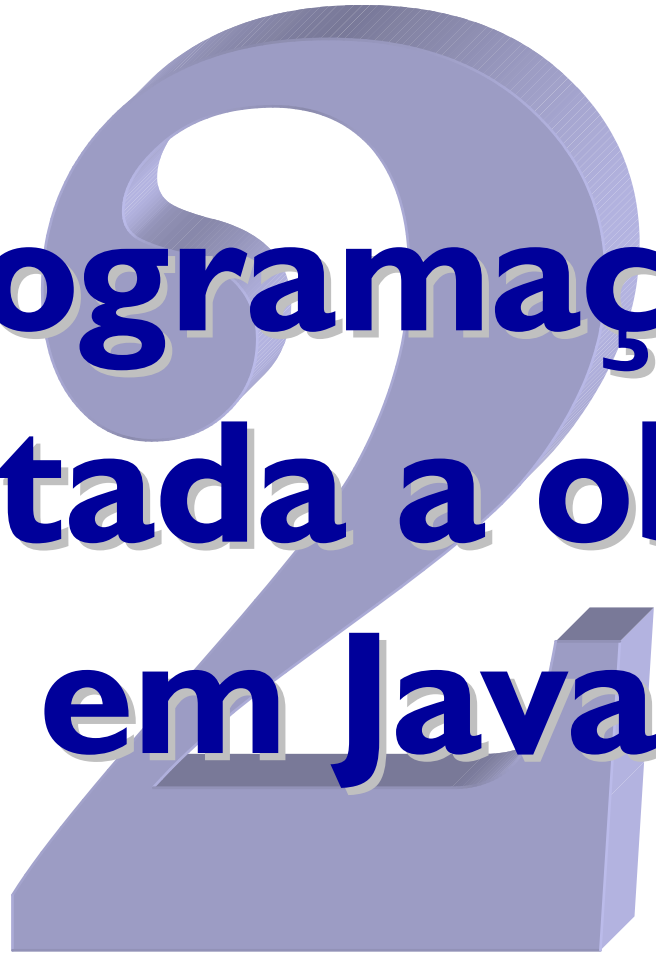


Programação orientada a objetos em Java

Helder da Rocha
www.argonavis.com.br

Assuntos abordados neste módulo

- Conceitos
  - pacote, classe, objeto, membro, atributo, método, construtor e interface
  - abstração, encapsulamento, herança e polimorfismo
- Construção de aplicações
  - Como construir uma classe contendo métodos, atributos e construtores
  - Como usar uma classe para construir objetos

Parte 1: Orientação a objetos

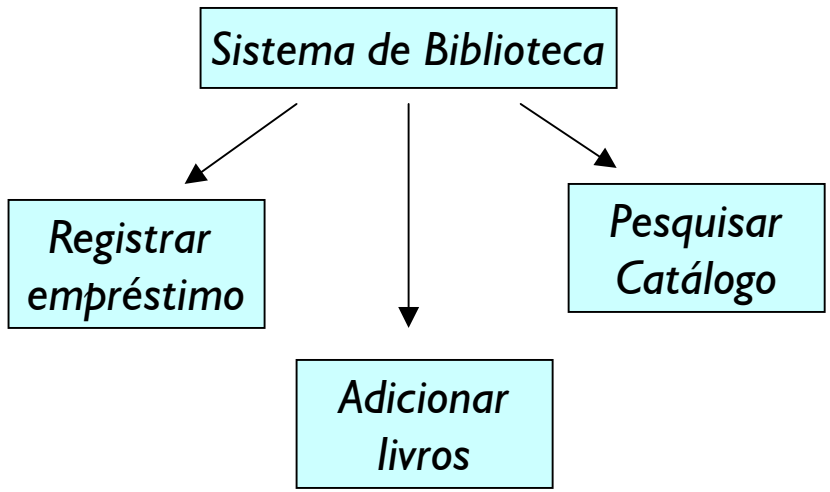
- *Paradigma do momento na engenharia de software*
- *A **análise** orientada a objetos*
  - *Determina **o que** o **sistema** deve fazer: Quais os atores envolvidos? Quais as atividades a serem realizadas?*
  - *Decompõe o sistema em **objetos**: Quais são? Que tarefas cada objeto terá que fazer?*
- *O **design** orientado a objetos*
  - *Define **como** o sistema será implementado*
  - *Modela os relacionamentos entre os objetos e atores*
    - *Pode-se usar uma linguagem específica como UML*
  - *Utiliza e reutiliza abstrações como classes, objetos, funções, frameworks e APIs*

A&D orientado a objeto vs. orientado a procedimentos

- *Formas de abstrair o problema (reduzir a sua complexidade para solucioná-lo): divide & conquer*

- *Trabalhar no **espaço do problema** (decompor em objetos)*
  - *Abstrações mais próximas do mundo real*

- *Trabalhar no **espaço da solução** (decompor em funções)*
  - *Abstrações mais próximas do mundo do computador*



O que é um objeto?

- Objetos são **conceitos** que têm
  - identidade,
  - estado e
  - comportamento
- Características de Smalltalk, resumidas por Allan Kay:
  - Tudo (em um programa OO) são objetos
  - Um programa é um monte de objetos enviando mensagens uns aos outros
  - O espaço (na memória) ocupado por um objeto consiste de outros objetos
  - Todo objeto possui um tipo (que descreve seus dados)
  - Objetos de um determinado tipo podem receber as mesmas mensagens

Objetos (2)

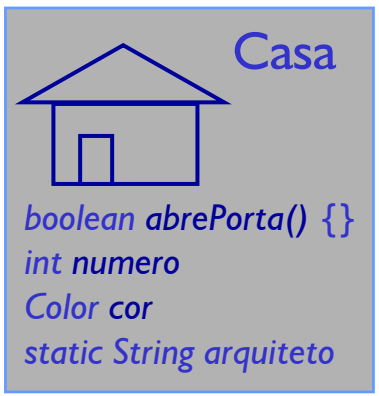
- Em uma linguagem orientada a objetos pura
  - Um número, uma letra, uma palavra, uma valor booleano, uma data, um registro, uma botão da GUI são objetos
- Em Java
  - Todos os objetos são armazenados no heap e manipulados através de uma referência (variável)
    - Têm estado (seus atributos)
    - Têm comportamento (seus métodos)
    - Têm identidade (a variável que contém sua referência)
  - Valores unidimensionais não são objetos
    - Números, booleanos, caracteres são armazenados na pilha
    - Têm apenas identidade (nome da variável) e estado (valor armazenado na variável)

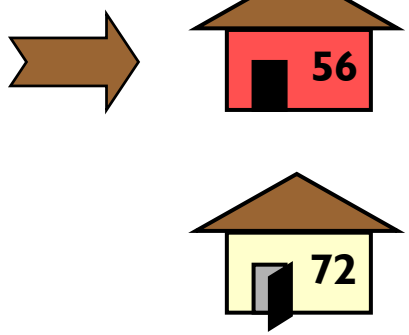
O que é uma classe?

- Classes são uma **especificação** para objetos
- Classes representam um **tipo de dados**
- Descrevem
  - Tipos dos dados que compõem o objeto (o que podem armazenar)
  - Funções que o objeto pode executar (o que podem fazer)

Instâncias da classe Casa (objetos)

Classe






```
Casa c1 = new Casa();
c1.numero = 12;
c1.cor = Color.yellow;

Casa c2 = new Casa();
c2.numero = 56;
c2.cor = Color.red;

Casa c3 = new Casa();
c3.numero = 72;
c3.cor = Color.white;
c3.abrePorta();
```

Membros: atributos e métodos

- *Uma classe é uma estrutura de dados*
  - *Pode conter construtores, métodos, atributos, blocos estáticos e classes internas em qualquer ordem*
- *Os componentes de uma classe são seus membros*
- *Há dois tipos de membros*
  - *membros estáticos*
  - *membros de instância*
- ***Membros estáticos***
  - *declarados com o modificador static (exceto construtores)*
  - *podem ser usados através da classe mesmo quando não há objetos*
  - *Não se replicam quando novos objetos são criados*
- ***Membros de instância***
  - *Cada objeto, quando criado, aloca espaço para eles*
  - *Só podem ser usados através de objetos*

Construtores

- ***Construtores*** *são procedimentos realizados na construção de objetos*
  - *Parecem métodos, mas não têm tipo de retorno e têm nome idêntico ao nome da classe*
  - *Para cada objeto, seu construtor é executado exatamente uma vez: na hora de sua criação*
- *Construtores podem ser chamados de duas formas*
  - *1. Através de operações de construção de objetos:*

    ```
    > Objeto obj = new Objeto();
    ```

  - *2. Através do construtor de uma subclasse*

Exemplo

- *Uma Casa é uma Construção*
  - *Tem um proprietario Humano, um número, uma rua e várias Portas*
  - *Pode abrir e fechar portas*
  - *Tem uma imobiliária (que já existia antes)*

```
public class Casa extends Construcao {
    private Humano proprietario;
    private Porta[] portas;
    private String rua;
    private int numero;
    public Casa() {
        proprietario = new Humano();
        portas = new Porta[2];
        rua = "Rua Esquerda";
        numero = contagem++ * 10;
    }
    public void abrePorta(int numero) {
        porta[numero].abre();
    }
    public static String imobiliaria = "XYZ Imóveis";
    private static int contagem = 1;
}
```

Atributos de instância (cada objeto tem uma cópia)

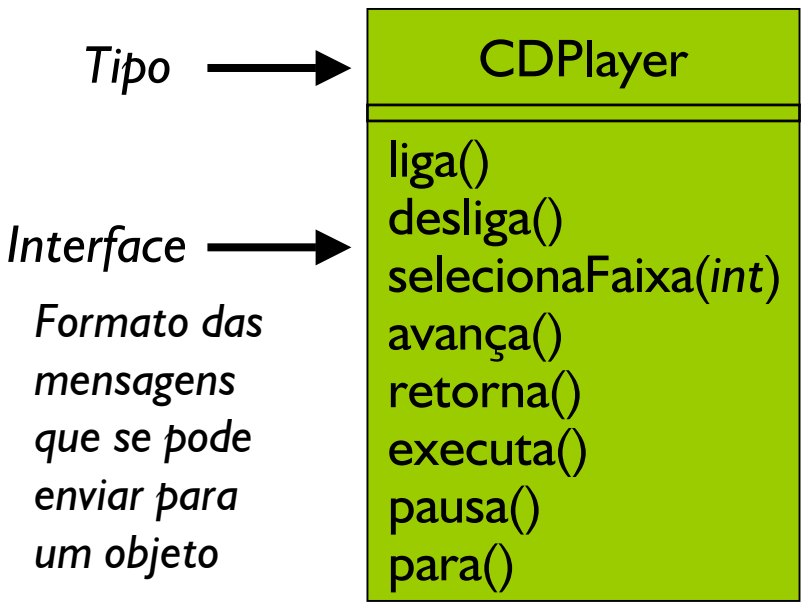
Construtor (chamado uma vez por objeto)

Método de instância (cada objeto tem um)

Campos estáticos (compartilhados)

Objetos possuem uma interface ...

- Através da **interface*** é possível utilizá-lo
  - Não é preciso saber dos detalhes da implementação
- O **tipo** (Classe) de um objeto determina sua interface
  - O tipo determina quais mensagens podem ser enviadas



Em Java

```
(...)
CDPlayer cd1;
cd1 = new CDPlayer();
cd1.liga();
cd1.selecionaFaixa(3);
cd1.executa();
(...)
```

Classe Java (tipo)

Referência

Criação de objeto

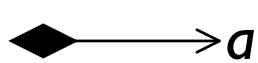
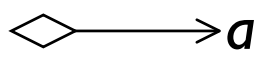
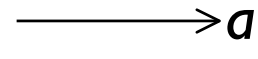
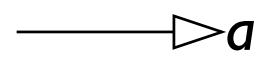
Envio de mensagem

* interface aqui refere-se a um conceito e não a um tipo de classe Java

... e uma implementação (oculta)

- Implementação não interessa à quem usa objetos
- Papel: usuário de classes
  - não precisa saber como a classe foi escrita, apenas quais seus métodos, quais os parâmetros (quantidade, ordem e tipo) e valores que são retornados
  - usa apenas a interface (pública) da classe
- Papel: desenvolvedor de classes
  - define novos tipos de dados
  - expõe, através de métodos, todas as funções necessárias ao usuário de classes, e oculta o resto da implementação
  - tem a liberdade de mudar a implementação das classes que cria sem que isto comprometa as aplicações desenvolvidas pelo usuário de classes

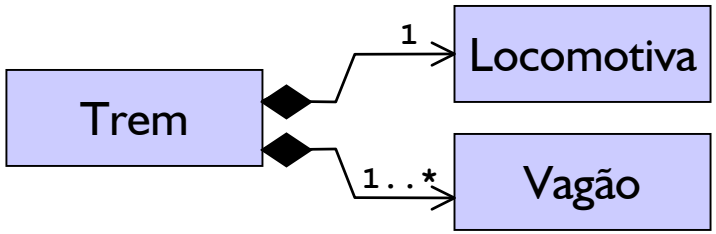
Reuso de implementação

- *Separação interface-implementação: maior reuso*
  - *Reuso depende de bom planejamento e design*
- *Uma vez criada uma classe, ela deve representar uma unidade de código útil para que seja reutilizável*
- *Formas de uso e reuso*
  - *Uso e reuso de **objetos** criados pela classe: mais flexível*
    - *Composição: a "é parte essencial de" b* 
    - *Agregação: a "é parte de" b* 
    - *Associação: a "é usado por" b* 
  - *Reuso da interface da **classe**: pouco flexível*
    - *Herança: b "é" a (substituição pura)* 

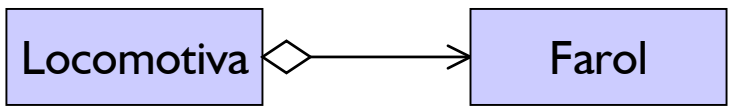
      ou b "é um tipo de" a (substituição útil, extensão)

Agregação, composição e associação

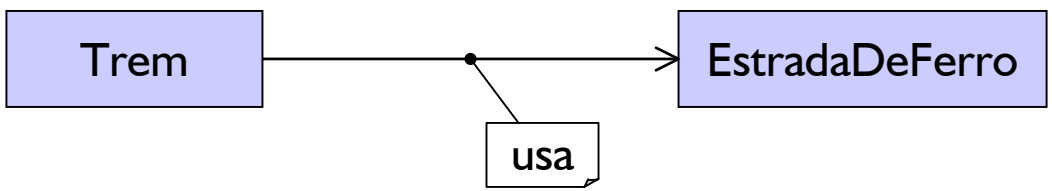
- ***Composição***: *um trem é formado por locomotiva e vagões*



- ***Agregação***: *uma locomotiva tem um farol (mas não vai deixar de ser uma locomotiva se não o tiver)*

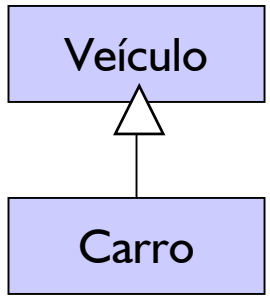


- ***Associação***: *um trem usa uma estrada de ferro (não faz parte do trem, mas ele depende dela)*



Herança (reuso de interface)

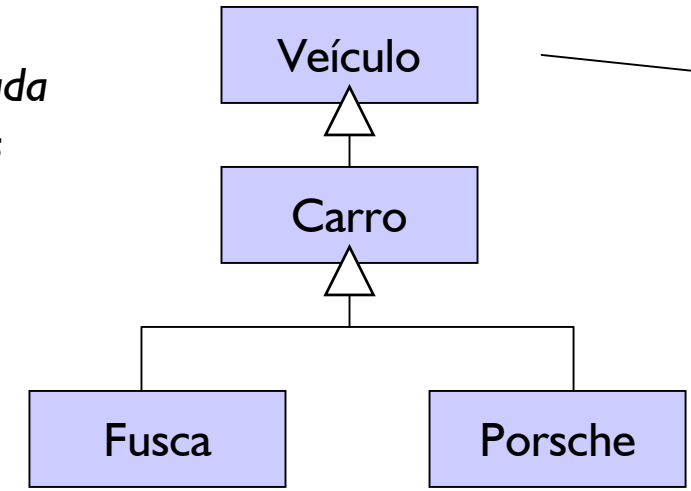
- *Um carro é um veículo*

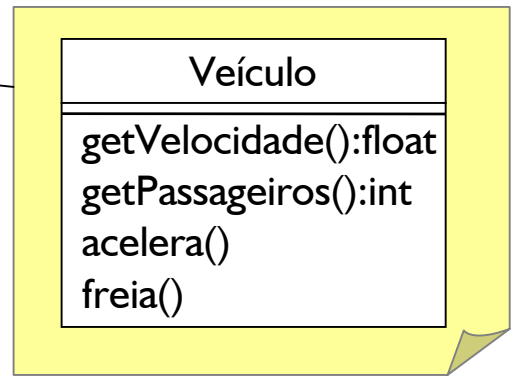


representação UML

- *Fuscas e Porsches são carros (e também veículos)*

representação UML simplificada (não mostra os métodos)

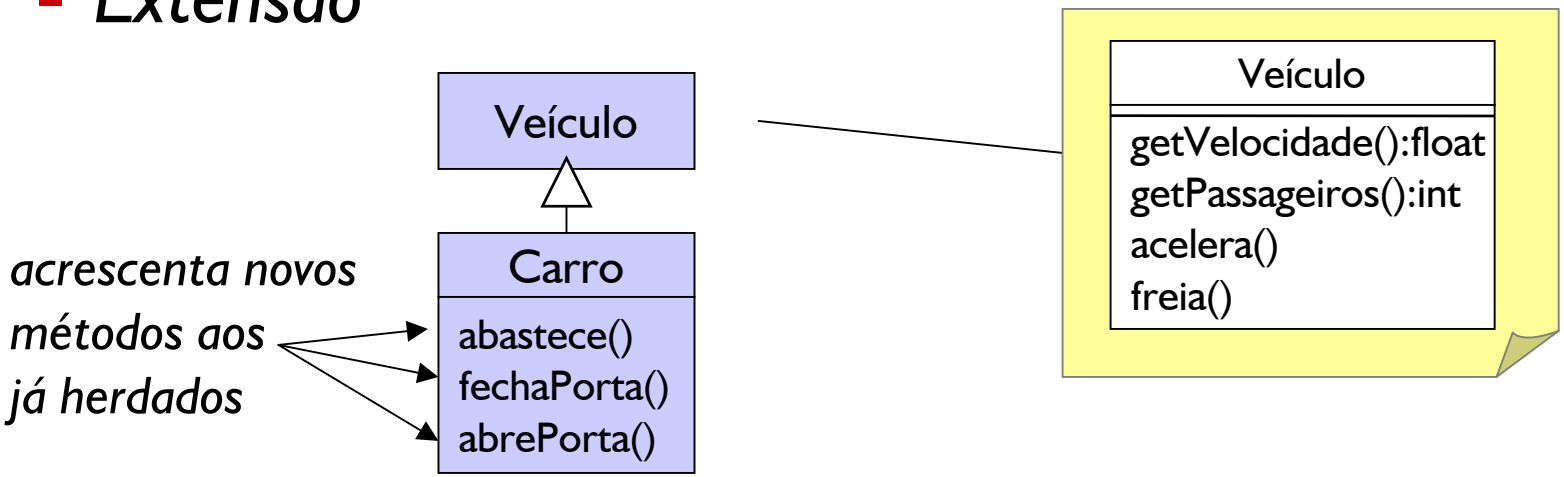




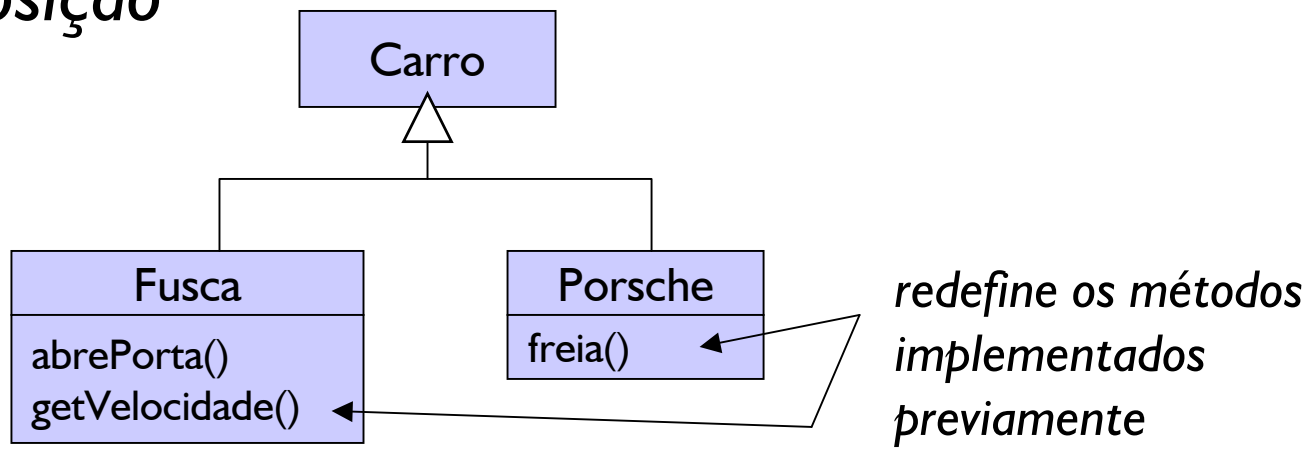
representação UML detalhada de 'Veículo'

Extensão e sobreposição

- *Extensão*

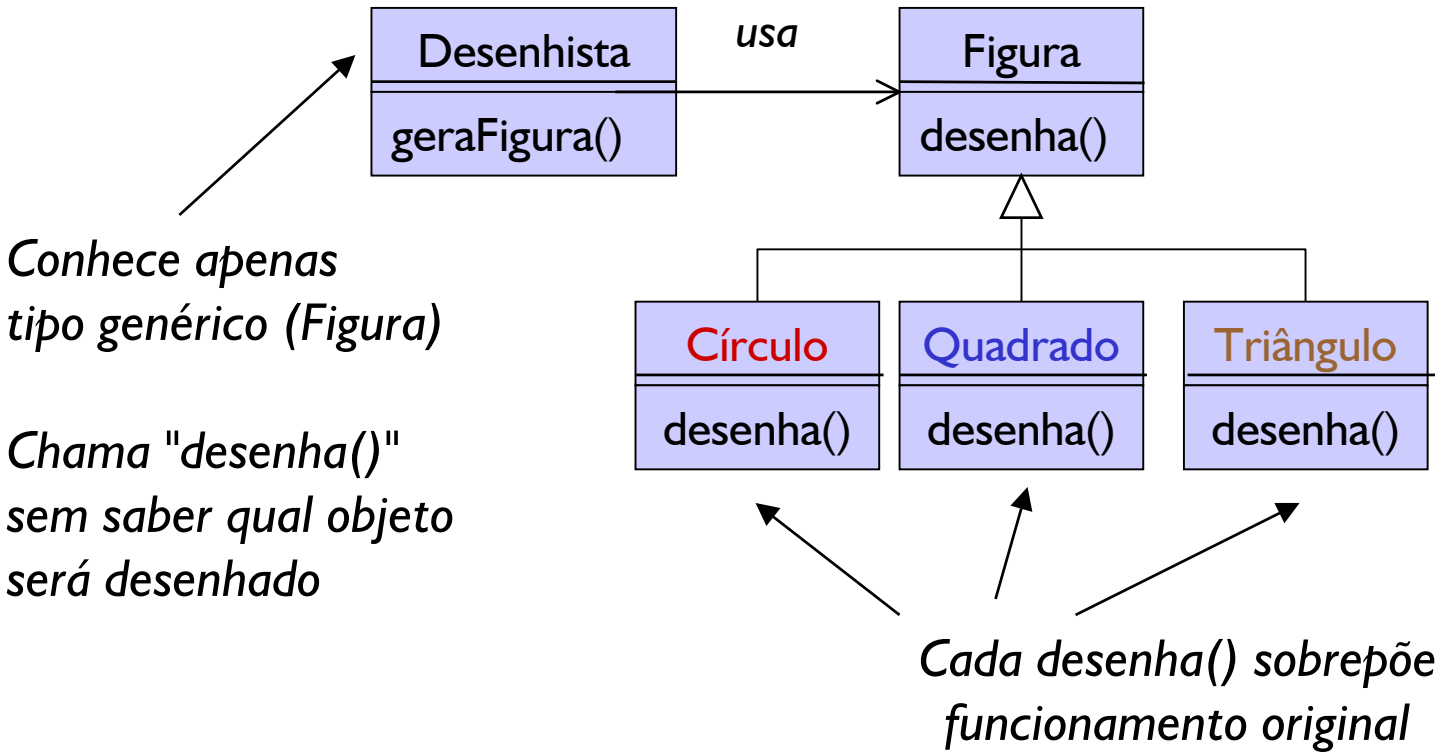


- *Sobreposição*



Polimorfismo

- *Uso de um objeto no lugar de outro*
  - *pode-se escrever código que não dependa da existência prévia de tipos específicos*



Encapsulamento

- Simplifica o objeto expondo apenas a sua interface essencial
- Código dentro de métodos é naturalmente encapsulado
  - Não é possível acessar interior de um método fora do objeto
- Métodos que não devem ser usados externamente e atributos podem ter seu nível de acesso controlado em Java, através de modificadores
  - **private**: apenas acesso dentro da classe
  - **package-private** (default): acesso dentro do pacote*
  - **protected**: acesso em subclasses
  - **public**: acesso global

---

* não existe um modificador com este nome. A ausência de um modificador de acesso deixa o membro com acesso package-private

Exercício

- 1. Crie as seguintes classes
  - Um Humano tem um nome (String)
  - Uma Porta tem um estado aberto, que pode ser true ou false, e pode ser aberta ou fechada
  - Uma Casa tem um proprietário Humano, um número e um conjunto de Portas
- 2. Crie as seguintes classes
  - Um Ponto tem coordenadas x e y inteiras
  - Um Circulo tem um Ponto e um raio inteiro

Tipos primitivos

- *Têm tamanho fixo. Têm sempre valor default.*
- *Armazenados na pilha (maior desempenho)*
- *Não são objetos. Classe 'wrapper ' faz transformação.*

Tipo	Tamanho	Mínimo	Máximo	'Wrapper'
boolean	—	—	—	**Boolean**
char	16-bit	Unicode 0	Unicode 2^{16} - 1	**Character**
byte	8-bit	-128	+127	**Byte**
short	16-bit	-2^{15}	$+2^{15}$ —1	**Short**
int	32-bit	-2^{31}	$+2^{31}$ —1	**Integer**
long	64-bit	-2^{63}	$+2^{63}$ —1	**Long**
float	32-bit	IEEE754	IEEE754	**Float**
double	64-bit	IEEE754	IEEE754	**Double**
void	—	—	—	**Void**

Tipos primitivos e literais

- Literais de caracter:
  - `char c = 'a';`
  - `char z = '\u0041'; // em Unicode`
- Literais de string
  - `String s = "abcde";  // ou String s = new String("abcde");`
- Literais inteiros
  - `int i = 10;  short s = 15;  byte b = 1;`
  - `long j = 0x9af0L; int k = 0633;`
- Literais de ponto-flutuante
  - `float f = 123.0f;`
  - `double d = 12.3;   double g = .1e-23;`
- Literais booleanos
  - `boolean v = true;    boolean f = false;`

Menor classe utilizável em Java

- Uma classe contém a representação de um objeto
  - define seus métodos (comportamento)
  - define os tipos de dados que o objeto pode armazenar (estado)
  - determina como o objeto deve ser criado (construtor)
- Uma classe Java também pode conter
  - procedimentos independentes (métodos 'static')
  - variáveis estáticas
  - rotinas de inicialização (blocos 'static')
- O programa abaixo é a menor unidade compilável em Java

```
class Menor {}
```

Símbolos essenciais

- Separadores
  - { ... } chaves
  - contém as partes de uma classe
  - delimitam blocos de instruções (em métodos, inicializadores, estruturas de controle, etc.)
  - ; ponto-e-vírgula: obrigatória no final de toda instrução simples ou declaração
- Identificadores
  - nomes usados para representar classes, métodos, variáveis (por exemplo: PrimeiraClasse, Casa, Humano)
  - Podem conter letras (Unicode) e números, mas não podem começar com número
- Palavras reservadas
  - são 49 (assert foi incluída na versão 1.4.0)
  - exemplos são 'public' e 'class'

Para que serve uma classe

- Uma classe pode ser usada para
  - conter a rotina de execução principal de uma aplicação iniciada pelo sistema operacional (método main)
  - conter funções globais (métodos estáticos)
  - conter constantes e variáveis globais (campos de dados estáticos)
  - especificar e criar objetos (contém construtores, métodos e atributos de dados)

Uma unidade de compilação

Casa.java

```
package cidade; // classe faz parte do pacote cidade

import cidade.ruas.*;    // usa todas as classes de pacote
import pais.terrenos.LoteUrbano; // usa classe LoteUrbano
import Pessoa;           // ilegal desde Java 1.4.0
import java.util.*;      // usa classes de pacote Java

class Garagem {
  ...
}

interface Fachada {
...
}

/** Classe principal */
public class Casa {
...
}
```

Por causa da declaração 'package' o nome completo destas classes é cidade.Garagem cidade.Fachada e cidade.Casa

Este arquivo, ao ser compilado, irá gerar três arquivos .class

O que pode conter uma classe

- Um bloco '*class*' pode conter (entre as chaves `{ . . . }`), em qualquer ordem
  - (1) zero ou mais declarações de **atributos de dados**
  - (2) zero ou mais definições de **métodos**
  - (3) zero ou mais **construtores**
  - (4) zero ou mais **blocos de inicialização static**
  - (5) zero ou mais definições de **classes ou interfaces internas**
- Esses elementos só podem ocorrer **dentro** do bloco '`class NomeDaClasse { . . . }`'
  - tudo, em Java, 'pertence' a alguma classe
  - apenas '*import*' e '*package*' podem ocorrer fora de uma declaração '*class*' (ou '*interface*')

Métodos

- *Contém procedimentos - instruções simples ou compostas executadas em seqüência - entre chaves*
- *Podem conter argumentos*
  - *O tipo de cada argumento precisa ser declarado*
  - *Método é identificado pelo nome + número e tipo de argumentos*
- *Possuem um tipo de retorno ou a palavra void*
- *Podem ter modificadores (public, static, etc.) antes do tipo*

```
...
  public void paint (Graphics g) {
     int x = 10;
     g.drawString(x, x*2, "Booo!");
  }
...
```

```
class T1 {
  private int a; private int b;
  public int soma () {
    return a + b;
  }
}
```

```
class T2 {
  int x, y;
  public int soma () {
    return x + y;
  }
  public static int soma (int a, int b) {
    return a + b;
  }

  public static int soma (int a, int b, int c) {
    return soma(soma(a, b), c);
  }
}
```

Sintaxe de definição de métodos

- *Sintaxe básica*
  - [mod]* *tipo identificador* ([tipo arg]*) [throws exceção*] { ... }
- *Chave*
  - [mod]* – *zero ou mais modificadores separados por espaços*
  - *tipo* – *tipo de dados retornado pelo método*
  - *identificador* – *nome do método*
  - [arg]* – *zero ou mais argumentos, com tipo declarado, separados por vírgula*
  - [throws exceção*] – *declaração de exceções*
- *Exemplos*
  - `public static void main ( String[] args ) { ... }`
  - `private final synchronized native int metodo (int i, int j, int k) ;`
  - `String abreArquivo () throws IOException, Excecao2 { ... }`

Atributos de dados

- *Contém dados*
- *Devem ser declaradas com tipo*
- *Podem ser pré-inicializadas (ou não)*
- *Podem conter modificadores*

```
public class Produto {
   public static int total = 0;
   public int serie = 0;
   public Produto() {
      serie = serie + 1;
      total = serie;
   }
}
```

```
class Data {
   int dia;
   int mes;
   int ano;
}
```

```
public class Livro {
   private String titulo;
   private int codigo = 815;
   ...
   public int mostraCodigo() {
      return codigo;
   }
}
```

```
class Casa {
   static Humano arquiteto;
   int numero;
   Humano proprietario;
   Doberman[] guardas;
}
```

Sintaxe de declaração de atributos

- *Sintaxe básica*
  - [modificador]* *tipo identificador* [= valor] ;
- *Chave*
  - [modificador]* – *zero ou mais modificadores (de acesso, de qualidade), separados por espaços*
    - *public, private, static, transient, final, etc.*
  - *tipo – tipo de dados que a variável (ou constante) pode conter*
  - *identificador – nome da variável ou constante*
  - [= valor] – *valor inicial da variável ou constante*
- *Exemplo*
  - `protected static final double PI = 3.14159 ;`
  - `int numero;`

Construtores

- *Têm sempre o mesmo nome que a classe*
- *Contém procedimentos entre chaves, como os métodos*
- *São chamados apenas uma vez: na criação do objeto*
- *Pode haver vários em uma mesma classe*
  - *são identificados pelo número e tipo de argumentos*
- *Nunca declaram tipo de retorno*

```
public class Produto {
   public static int total = 0;
   public int serie = 0;
   public Produto() {
      serie = serie + 1;
      total = serie;
   }
}
```

```
public class Livro {
   private String titulo;
   public Livro() {
      titulo = "Sem título";
   }
   public Livro(String umTitulo) {
      titulo = umTitulo;
   }
}
```

Sintaxe de construtores

- *Construtores são procedimentos especiais usados para construir novos objetos a partir de uma classe*
  - *A definição de construtores é opcional*
    - *toda classe sem construtor declarado explicitamente possui um construtor fornecido pelo sistema (sem argumentos)*
- *Parecem métodos mas*
  - *não definem tipo de retorno*
  - *possuem, como identificador, o nome da classe*
    - *Uma classe pode ter vários construtores, com o mesmo nome, que se distinguem pelo número e tipo de argumentos*
- *Sintaxe*
  - [mod]* *nome_classe* ([tipo arg]*) [throws exceção*] { ... }

Outros componentes

- *São usados menos freqüentemente*
  - ***(4) Blocos 'static'** servem para inicializar uma classe*
    - *serão abordados em módulo futuro*

```
static {
    ... instruções ...
}
```

  - ***(5) Classes internas** são definições de classes contidas dentro de outras classes e dentro de métodos*
    - *serão abordadas em módulo futuro*

Exemplo

- *Exemplo de classe com um atributo de dados (variável), um construtor e dois métodos*

```
public class PrimeiraClasse {

    private String mensagem;

    public PrimeiraClasse () {
        mensagem = "Mensagem inicial";
    }

    public void setMensagem (String m) {
        mensagem = m;
    }

    public String getMensagem() {
        return mensagem;
    }
}
```

variavel (referencia) do tipo String

construtor

inicialização de variável ocorre quando objeto é construído

método que recebe parâmetro e altera variável

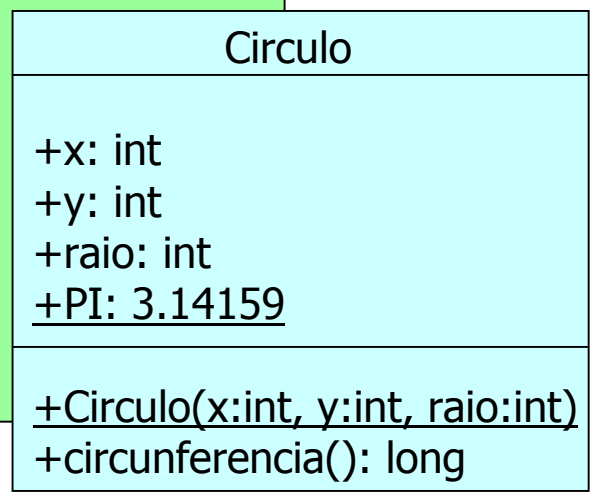
método que retorna variável

Exemplo: um círculo

```
public class Circulo {
   public int x;
   public int y;
   public int raio;
   public static final PI = 3.14159;

   public Circulo (int x1, int y1, int r) {
      x = x1;
      y = y1;
      raio = r;
   }

   public long circunferencia() {
      return 2 * PI * raio;
   }
}
```



Usando círculos

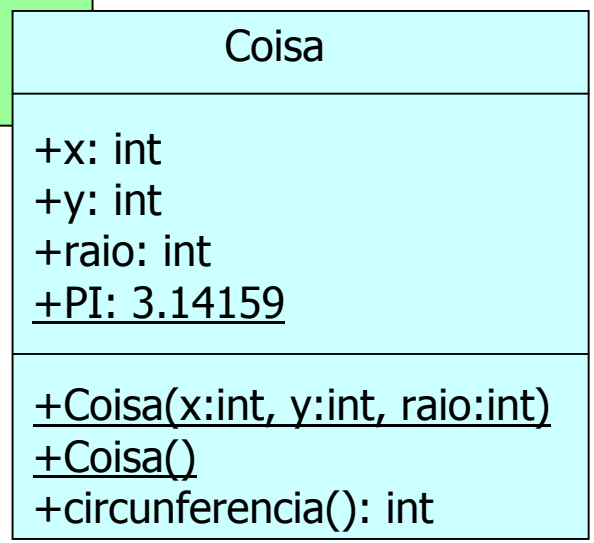
- *Use dentro de um método ou construtor (blocos de procedimentos)*

```
Circulo c1, c2, c3;
c1 = new Circulo(3, 3, 1);
c2 = new Circulo(2, 1, 4);
c3 = c1; // mesmo objeto!
System.out.println("c1: (" + c1.x + ", "
                           + c1.y + ")");
int circ = c1.circunferencia();
System.out.print("Raio de c1: " + c1.raio);
System.out.println("; Circunferência de c1: "
                   + circ);
```

Herança

```
class Coisa extends Circulo {
    Coisa() {
        this(1, 1, 0);
    }
    Coisa(int x, int y, int z) {
        super(x, y, z);
    }
}
```

A Coisa **é um** Circulo!



Exercício

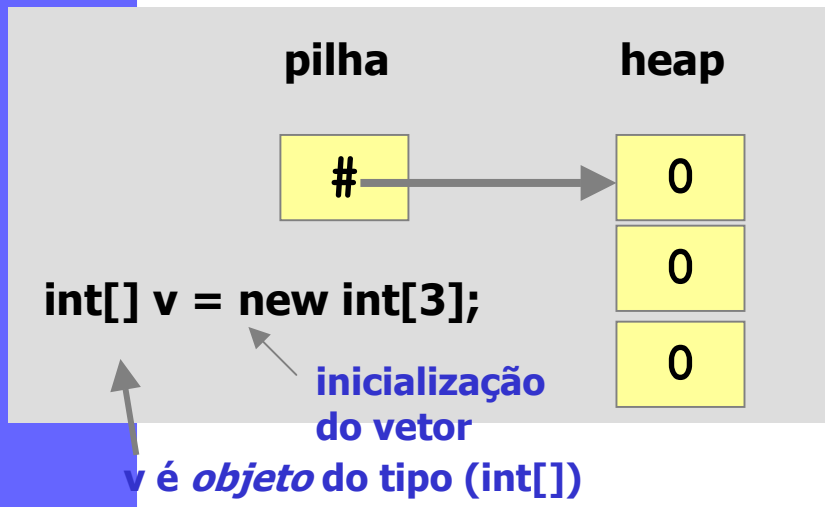
- 1. Escreva uma classe Ponto
  - contém x e y que podem ser definidos em construtor
  - métodos getX() e getY() que retornam x e y
  - métodos setX(int) e setY(int) que mudam x e y
- 2. Escreva uma classe Circulo, que contenha
  - raio inteiro e origem Ponto
  - construtor que define origem e raio
  - método que retorna a área
  - método que retorna a circunferência
  - use java.lang.Math.PI (Math.PI)
- 3. Crie um segundo construtor para Circulo que aceite
  - um raio do tipo int e coordenadas x e y

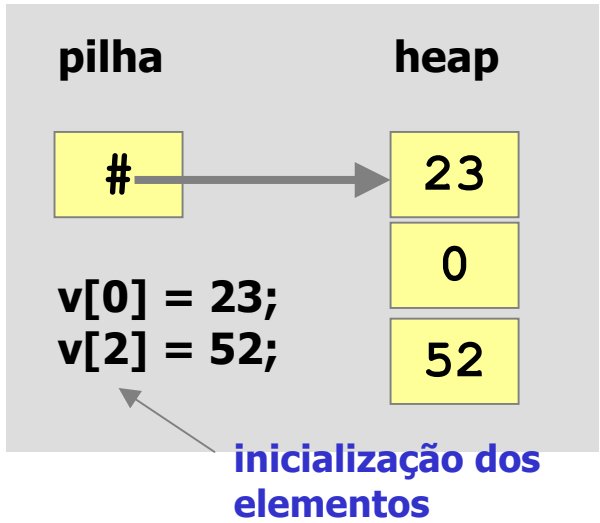
Vetores

- Vetores são coleções de objetos ou tipos primitivos
  - Os tipos devem ser conversíveis ao tipo em que foi declarado o vetor
  - `int[] vetor = new int[10];`
- Cada elemento do vetor é inicializado a um valor default, dependendo do tipo de dados:
  - null, para objetos
  - 0, para int, long, short, byte, float, double
  - Unicode 0, para char
  - false, para boolean
- Elementos podem ser recuperados a partir da posição 0:
  - `int elemento_1 = vetor[0];`
  - `int elemento_2 = vetor[1];`

Vetores

- *de tipos primitivos*

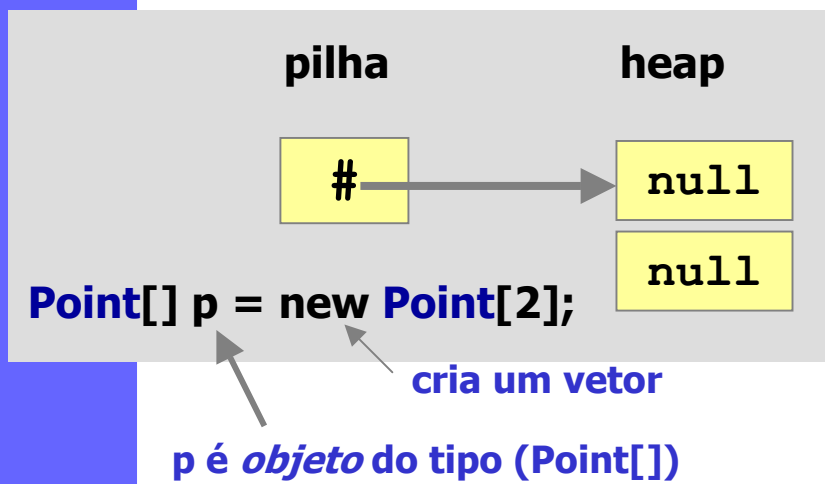


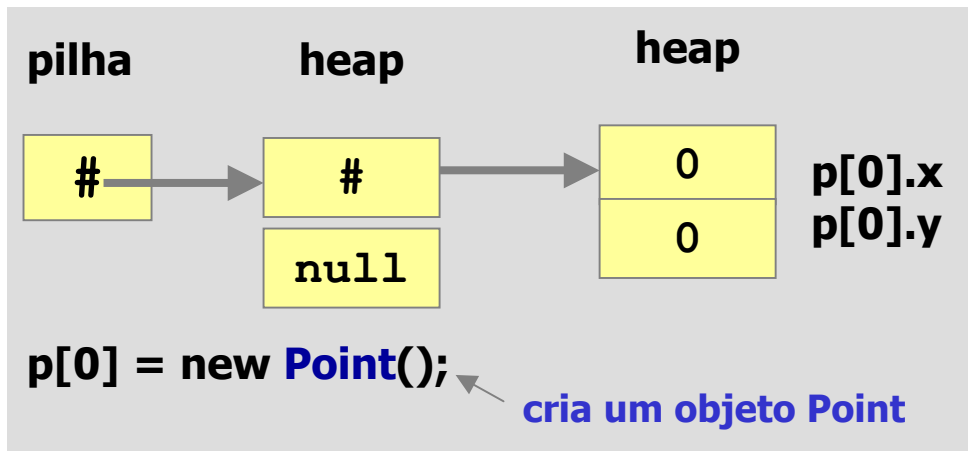



```
class Point {
   public int x;
   public int y;
}
```

Point
+x: int +y: int

- *de objetos (Point é uma classe, com dois membros x e y, inteiros)*





Inicialização de vetores

- *Vetores podem ser inicializados no momento em que são criados. Sintaxe:*
  - ```
    String[] semana = {"Dom", "Seg", "Ter",
                       "Qua", "Qui", "Sex","Sab"};
    ```
  - ```
    String[][] usuarios = {
        {"João", "Ninguém"},
        {"Maria", "D.", "Aparecida"},
        {"Fulano", "de", "Tal"}
    };
    ```
- *Essa inicialização não pode ser usada em outras situações (depois que o vetor já existe), exceto usando new, da forma:*
  - ```
    semana =
      new String[] {"Dom", "Seg", "Ter", "Qua",
                    "Qui", "Sex", "Sab"};
    ```
    ```

A propriedade length

- *Uma vez criados, vetores não podem ser redimensionados*
  - *Use **System.arraycopy()** para copiar um vetor para dentro de outro (alto desempenho)*
  - *Use **java.util.ArrayList** (ou Vector) para manipular com vetores de tamanho variável (baixo desempenho)*
  - *ArrayLists e Vectors são facilmente conversíveis em vetores quando necessário*
- ***length**: todo vetor em Java possui esta propriedade que informa o número de elementos que possui*
  - *length é uma propriedade read-only*
  - *extremamente útil em blocos de repetição*

```
for (int x = 0; x < vetor.length; x++) {
   vetor[x] = x*x;
}
```

Vetores multidimensionais

- *Vetores multidimensionais em Java são vetores de vetores*
  - *É possível criar toda a hierarquia (vetor de vetor de vetor...), para fazer vetores retangulares ...*

```
int [][][] prisma = new int [3][2][2];
```

  - *... ou criar apenas o primeiro nível (antes de usar, porém, é preciso criar os outros níveis)*

```
int [][][] prisma2 = new int [3][][];
prisma[0] = new int[2][];
prisma[1] = new int[3][2];
prisma[2] = new int[4][4];
prisma[0][0] = new int[5];
prisma[0][1] = new int[3];
```

Exercícios

- *1. Crie uma classe TestaCirculos que*
  - *a) crie um vetor de 5 objetos Circulo*
  - *b) imprima os valores x, y, raio de cada objeto*
  - *c) declare outra referência do tipo Circulo[]*
  - *d) copie a referência do primeiro vetor para o segundo*
  - *e) imprima ambos os vetores*
  - *f) crie um terceiro vetor*
  - *g) copie os objetos do primeiro vetor para o terceiro*
  - *h) altere os valores de raio para os objetos do primeiro vetor*
  - *i) imprima os três vetores*

Escopo de variáveis

- *Campos de dados (declarados no bloco da classe): podem ser usadas em qualquer lugar (qualquer bloco) da classe*
  - *Uso em outras classes depende de modificadores de acesso (public, private, etc.)*
  - *Existem enquanto o objeto existir (ou enquanto a classe existir, se declarados static)*
- *Variáveis locais (declaradas dentro de blocos de procedimentos)*
  - *Existem enquanto procedimento (método, bloco de controle de execução) estiver sendo executado*
  - *Não podem ser usadas fora do bloco*
  - *Não pode ter modificadores de acesso (private, public, etc.)*

Exemplo

variáveis visíveis dentro da classe, apenas

novoRaio *é variável local ao método* **mudaRaio**

maxRaio *é variável local ao método* **mudaRaio**

raio *é variável de instância*

inutil *é variável local ao bloco* **if**

```
public class Circulo {

  private int raio;
  private int x, y;

  public double area() {
    return Math.PI * raio * raio;
  }

  public void mudaRaio(int novoRaio) {
    int maxRaio = 50;
    if (novoRaio > maxRaio) {
      raio = maxRaio;
    }
    if (novoRaio > 0) {
      int inutil = 0;
      raio = novoRaio;
    }
  }
}
```

Membros de instância vs. componentes estáticos (de classe)

- Componentes estáticos
  - Os componentes de uma classe, quando declarados 'static', existem independente da criação de objetos
  - Só existe uma cópia de cada variável ou método
- Membros de instância
  - métodos e variáveis que **não tenham** modificador 'static' são membros do objeto
  - Para cada objeto, há uma cópia dos métodos e variáveis
- Escopo
  - Membros de instância não podem ser usados dentro de blocos estáticos
    - É preciso obter antes, uma referência para o objeto

Exemplos

- *Membros de instância só existem se houver um objeto*

main() ***não faz parte do objeto!***

Circulo
+**x**: int +**y**: int +**raio**: int

Errado!

```
public class Circulo {

   public int raio;
   public int x, y;

   public double area() {
      return Math.PI * raio * raio;
   }

   public static void main(String[] args) {
      raio = 3;
      double z = area();
   }
}
```

membros de instância

Pode. Porque area() *faz parte do objeto!*

qual *raio? existe?*

qual *area? existe?*

Não pode. Não existe objeto!

Certo!

```
public class Circulo {

   public int raio;
   public int x, y;

   public double area() {
      return Math.PI * raio * raio;
   }

   public static void main(String[] args) {
      Circulo c = new Circulo();
      c.raio = 3;
      double z = c.area();
   }
}
```

tem que criar pelo menos um objeto!

raio de c

area() de c

Variáveis locais vs. variáveis de instância

- *Variáveis de instância ...*
  - *sempre são automaticamente inicializadas*
  - *são sempre disponíveis no interior dos métodos de instância e construtores*
- *Variáveis locais ...*
  - *sempre têm que ser inicializadas antes do uso*
  - *podem ter o mesmo identificador que variáveis de instância*
  - *neste caso, é preciso usar a palavra reservada* **this** *para fazer a distinção*

```
class Circulo {
    private int raio;
    public void mudaRaio(int raio) {
        int x;
        raio = x;
        this.raio = raio;
    }
}
```

variável de instância

variável local

Comentários

- Há duas formas de incluir comentários em um arquivo Java
  - /* ... comentário de bloco ... */
  - // comentário de linha
- Antes de métodos, construtores, campos de dados e classes, o comentário de bloco iniciado com /** pode ser usado para gerar HTML em documentação
  - Há uma ferramenta (JavaDoc) que gera automaticamente documentação a partir dos arquivos .java
  - relaciona e descreve classes, métodos, etc e cria referências cruzadas
  - Descrições em HTML podem ser incluídas nos comentários especiais /** ... */

Geração de documentação

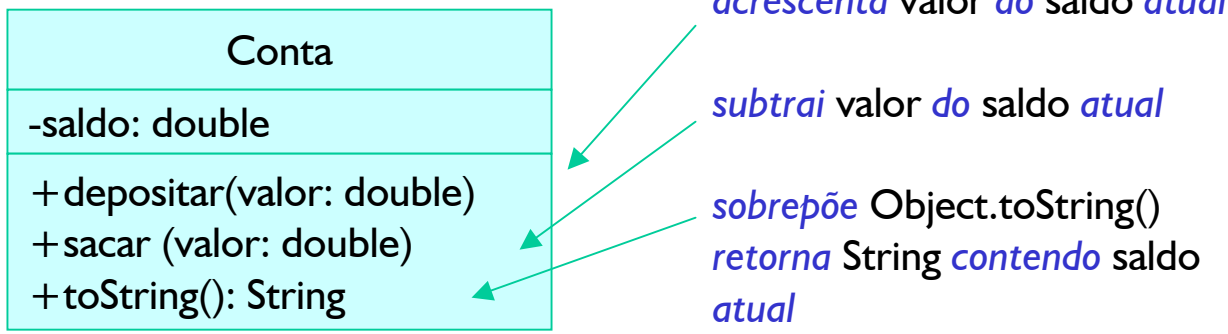
- *Para gerar documentação de um arquivo ou de uma coleção de arquivos .java use o javadoc:*
  - `javadoc arquivo1.java arquivo2.java`
- *O programa criará uma coleção de arquivos HTML, interligados, entre eles estarão*
  - *índice de referências cruzadas*
  - *uma página para cada classe, com links para cada método, construtor e campo público, contendo descrições (se houver) de comentários /** .. */*
- *Consulte a documentação para maiores informações sobre a ferramenta javadoc.*

Convenções de código

- Toda a documentação Java usa uma convenção para nomes de classes, métodos e variáveis
  - Utilizá-la facilitará a manutenção do seu código!
- Classes, construtores e interfaces
  - use caixa-mista com primeira letra maiúscula, iniciando novas palavras com caixa-alta. Não use sublinhado.
  - ex: UmaClasse, Livro
- Métodos e variáveis
  - use caixa mista, com primeira letra minúscula
  - ex: umaVariavel, umMetodo()
- Constantes
  - use todas as letras maiúsculas. Use sublinhado para separar as palavras
  - ex: UMA_CONSTANTE

Exercícios

- *1. Classe Conta e TestaConta*
  - *a) Crie a classe **Conta**, de acordo com o diagrama UML abaixo*



  - *b) Crie uma classe **TestaConta**, contendo um método main(), e simule a criação de objetos Conta, o uso dos métodos depositar() e sacar() e imprima, após cada operação, os valores disponíveis através do método toString()*
  - *c) Gere a documentação javadoc das duas classes*