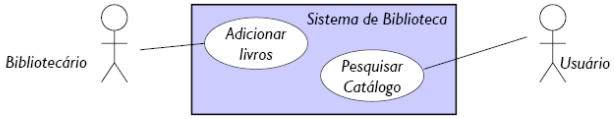
ORIENTAÇÃO A OBJETOS

Mário Meireles Teixeira
mario@deinf.ufma.br

O que é Orientação a Objetos

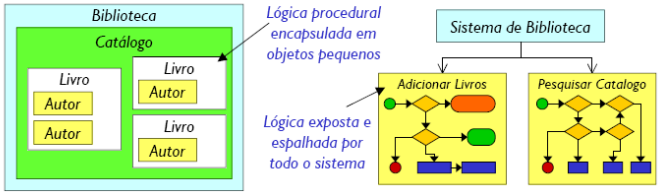
- Paradigma “moderno” da engenharia de software
  - Influi na análise, projeto (design) e programação
- A **análise** orientada a objetos
  - Determina **o que** o **sistema** deve fazer: Quais os atores envolvidos? Quais as atividades a serem realizadas?
  - Decompõe o sistema em **objetos**: Quais 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, métodos, frameworks, APIs, padrões de projeto

Visão OO (1) e Visão Procedural (2)



(1) Trabalha no **espaço do problema** (casos de uso simplificados em objetos)
- Abstrações mais simples e mais próximas do mundo real

(2) Trabalha no **espaço da solução** (casos de uso decompostos em procedimentos algorítmicos)
- Abstrações mais próximas do mundo do computador



O que é um objeto?

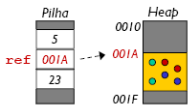
- Um objeto é uma entidade (física, conceitual ou do domínio de algum problema) que tem:
  - identidade
  - estado e
  - comportamento
- Características de Smalltalk, resumidas por Allan Kay:
  - Tudo (em um programa OO) são objetos
  - Um programa é um conjunto de objetos enviando mensagens uns aos 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

- Em uma linguagem OO pura
  - Uma variável é um objeto
  - Um programa é um objeto
  - Um procedimento é um objeto
  - Um objeto é composto de objetos, portanto
  - Um programa (objeto) pode ter variáveis (objetos que representam seu estado) e procedimentos (objetos que representam seu comportamento)
- Analogia: abstração de um telefone celular
  - É composto de outros objetos, entre eles bateria e botões
  - A bateria é um objeto também, que possui pelo menos um outro objeto: carga, que representa seu estado
  - Os botões implementam comportamentos

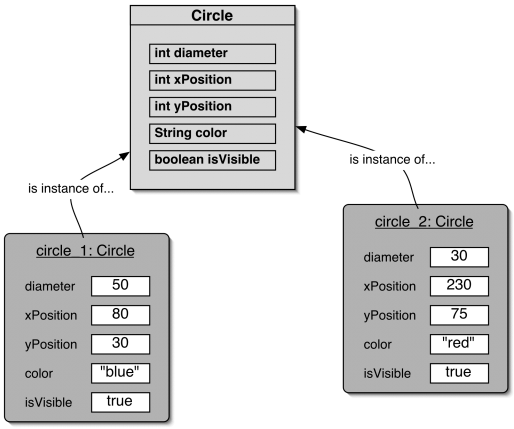
Objetos

- Em uma linguagem orientada a objetos pura
  - Um número, uma letra, uma palavra, uma valor booleano, uma data, um registro, um botão da interface são objetos
- Em Java, objetos são armazenados na memória de heap e manipulados através de uma referência (variável), guardada na pilha
  - Têm **identidade** (a referência)
  - Têm **estado** (seus atributos)
  - Têm **comportamento** (seus métodos)



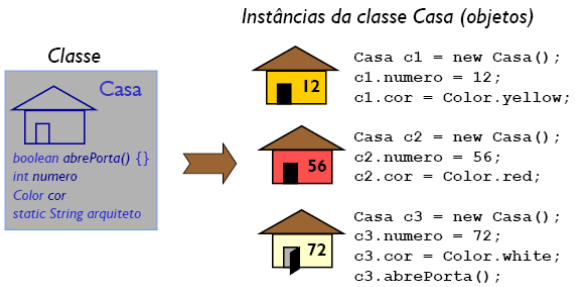
- Valores unidimensionais (tipos primitivos) não são objetos em Java
  - Números, booleanos, caracteres são armazenados na pilha
  - Têm apenas identidade (nome da variável) e estado (valor literal armazenado na variável); - dinâmicos; + rápidos

Exemplo: Dois Objetos Círculo



O que é uma classe?

- Classes são uma especificação para objetos com propriedades semelhantes (atributos), comportamentos semelhantes (métodos) e relacionamentos comuns com outros objetos
- Uma classe representa um tipo de dados complexo
- Classes descrevem
  - Tipos dos dados que compõem o objeto (o que podem armazenar)
  - Procedimentos que o objeto pode executar (o que podem fazer)



Classes

- Classes definem lógica estática
  - Relacionamentos entre classes são estáticos, definidos em tempo de programação e não mudam durante a execução
  - Relacionamentos entre objetos são dinâmicos e podem mudar. O funcionamento da aplicação reflete a lógica de relacionamento entre os objetos, e não entre as classes
- Classes não existem no contexto de execução
  - Uma classe representa vários objetos que ocupam espaço na memória, mas ela não existe nesse domínio
  - A classe tem papel na criação dos objetos, mas as mensagens são trocadas entre os objetos
- A classe é a "planta", o objeto é a "casa" construída. Muitas casas podem ser feitas a partir da mesma planta, mas cada uma preserva sua própria identidade

Definição de classe em Java

```
class Conta {
    String numero;
    double saldo;
    void creditar (double valor) {
        saldo = saldo + valor;
    }
    void debitar (double valor) {
        saldo = saldo - valor;
    }
}
```

Atributos

Métodos

Exemplo: Programa que cria e manipula Conta

```
class  CriaConta {
  /* Criando um objeto do tipo Conta  */
  public static void main (String [] args) {
    Conta conta1 = new Conta ();
    conta1.numero = "21.342-7";
    conta1.saldo = 0;
    conta1.creditar (500.87);
    conta1.debitar (45.00);
    System.out.println(conta1.saldo);
  }
}
```


Membros de uma classe: atributos e métodos

- Uma classe define uma estrutura de dados não-ordenada, podendo conter componentes em qualquer ordem
- Os componentes de uma classe são seus membros
- Uma classe pode conter três tipos de componentes
  - Membros estáticos ou de classe: não fazem parte do "tipo" do objeto
  - Membros de instância: definem o tipo de um objeto
  - Procedimentos de inicialização

- Membros estáticos ou de classe
  - Podem ser usados através da classe mesmo quando não há objetos
  - Não se replicam quando novos objetos são criados ("variáveis globais" da classe)
- Membros de instância
  - Cada objeto, quando criado, aloca espaço para eles
  - Só podem ser usados através de objetos
- Procedimentos de inicialização (construtores)
  - Usados para inicializar objetos ou classes

Usando membros estáticos

- Classes podem declarar membros (campos e métodos) que sejam comuns a todas as instâncias, ou seja, membros compartilhados por todos os objetos da classe
- Tais membros são comumente chamados de 'membros de classe' (versus 'de objetos')
- Em Java, declara-se um membro de classe usando o qualificador **static.** Daí, o nome 'membros estáticos' usado em Java.

```
class Conta {
   static String nomeBanco = "Itaú";
   String numero;
   double saldo;
   ...
   void creditar(double valor) {saldo += valor; }
   void debitar(double valor) {saldo -= valor; }
}
```


Construtores

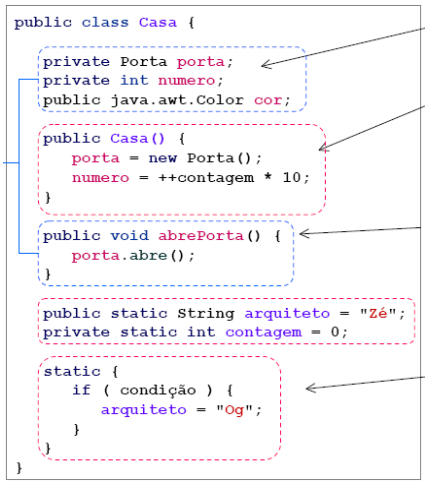
- São procedimentos executados na criação do objeto, uma única vez
- Têm o mesmo nome da classe. São similares a métodos, mas não têm tipo de retorno. Não fazem parte da definição do tipo do objeto (interface)

```
class Conta {
   String numero;
   double saldo;
   Conta(String n) {
      numero = n;
      saldo = 0;
   }
   void creditar(double valor) {saldo += valor; }
   void debitar(double valor) {saldo -= valor; }
}
```

Criando Objetos com Construtores

```
. . .
Conta  conta1;
conta1 = new Conta("21.342-7");
conta1.creditar(500.87);
conta1.debitar(45.00);
System.out.println(conta1.saldo);
. . .
```


Terminologia



Atributos de instância: cada objeto poderá armazenar valores diferentes nessas variáveis.

Procedimento de inicialização de objetos (Construtor): código é executado após a criação de cada novo objeto. Cada objeto terá um número diferente.

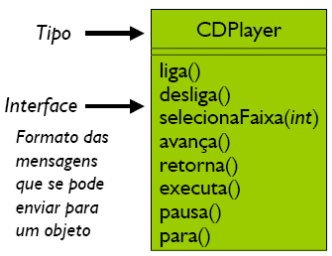
Método de instância: só é possível chamá-lo se for através de um objeto.

Atributos estáticos: não é preciso criar objetos para usá-los. Todos os objetos os compartilham.

Procedimento de inicialização estático: código é executado uma única vez, quando a classe é carregada. O arquiteto será um só para todas as casas: ou Zé ou Og.

Objetos possuem uma interface...

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



Em Java

```
(...)                    Classe Java (tipo)
CDPlayer cd1;            Referência
cd1 = new CDPlayer();
cd1.liga();              Criação de objeto
cd1.selecionaFaixa(3);
cd1.executa();
(...)                    Envio de mensagem
```

Ao interagir com um objeto, levamos em conta a interface (o serviço) e não a implementação. Chamamos a isso de Abstração.

...e uma implementação (oculta)

- Implementação não interessa a quem **usa** o objeto
- Papel do usuário da classe:
  - não precisa saber como a classe foi escrita, apenas o nome dos métodos, seus parâmetros (quantidade, ordem e tipo) e valor de retorno – **assinatura** do método
  - usa apenas a interface (pública) da classe
- Papel do desenvolvedor da classe:
  - define novos tipos de dados
  - expõe, através de métodos, todas as funções necessárias ao usuário da classe, e oculta o resto da implementação
  - tem a liberdade de mudar a implementação da classe, desde que isso não comprometa sua interface

Resumo

- Os componentes de uma classe, em Java, podem pertencer a dois domínios, que determinam como os mesmos serão usados
  - Domínio da classe: existem independentemente de existirem objetos ou não – métodos static, blocos static, atributos static e interface dos construtores de objetos
  - Domínio do objeto: métodos e atributos não declarados como static (definem o tipo ou interface que um objeto possui), e conteúdo dos construtores
- Construtores são usados apenas para construir objetos
  - Não são métodos (não declaram tipo de retorno)
  - “Ponte" entre dois domínios: são chamados uma vez antes do objeto existir (domínio da classe) e executados no domínio do objeto criado
- Separação de interface e implementação
  - Usuários de classes vêem apenas a interface.
  - Implementação é encapsulada dentro dos métodos, e pode variar sem afetar as classes que usam os objetos

Desenvolvendo um exemplo

Uma máquina de venda de bilhetes

Máquina de venda de bilhetes: visão externa

- Explorando o comportamento de uma máquina de venda de bilhetes:
  - A máquina fornece bilhetes de preço fixo
  - Os clientes inserem dinheiro na máquina e depois solicitam que um bilhete seja impresso
  - A máquina mantém um total geral do dinheiro coletado durante sua operação

Estrutura de uma classe básica em Java

```
public class TicketMachine
{
    Parte interna da classe omitida.
}
```

O empacotador externo da TicketMachine

```
public class NomeDaClasse
{
    Campos/Atributos
    Construtores
    Métodos
}
```

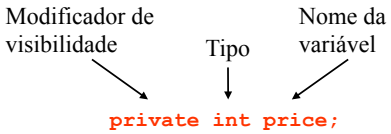
O conteúdo de uma classe

Atributos ou campos

- Atributos armazenam valores para um objeto
- Eles também são conhecidos como variáveis de instância
- Atributos definem o estado de um objeto

```
Public class TicketMachine
{
    private int price;
    private int balance;
    private int total;

    Construtor e métodos omitidos
}
```



Métodos

```
public class TicketMachine
{
  Atributos omitidos

  public TicketMachine(int ticketCost){ }

   public int getPrice(){ return price; }

   public int getBalance(){ return balance; }

   public void insertMoney(int amount) { }

   public void printTicket(){ }

}
```

Construtores

- Construtores inicializam um objeto
- Têm o mesmo nome de sua classe
- Inicializam os atributos
- Freqüentemente recebem valores de parâmetros externos à classe

```
public class TicketMachine
{

  public TicketMachine(int ticketCost)
  {
      price = ticketCost;
      balance = 0;
      total = 0;
  }

}
```

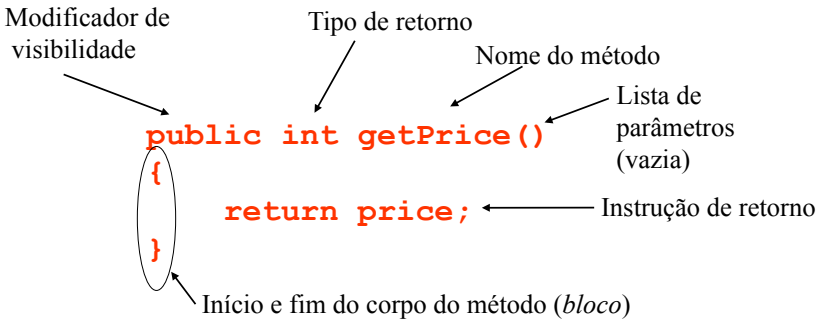

Métodos de acesso

- Métodos implementam o comportamento dos objetos
- Métodos de acesso fornecem informações sobre um objeto
- A classe `TicketMachine` possui os seguintes métodos:
  - **`getPrice`**, **`getBalance`**, `insertMoney`, `printTicket`
- Métodos têm uma estrutura que consiste em um cabeçalho e um corpo
- O cabeçalho define a *assinatura do* método:

```
public int getPrice()
```

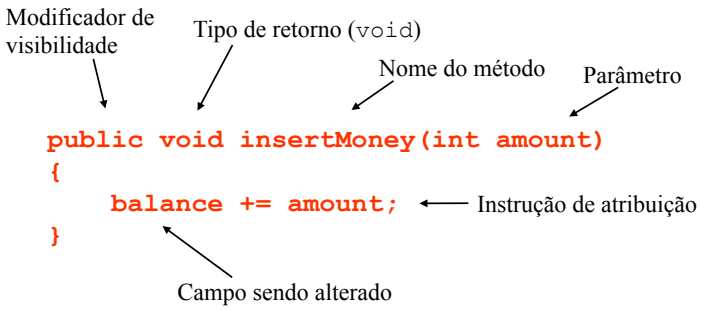
- O corpo engloba as instruções do método

Métodos de acesso (*getters*)



Métodos modificadores (*setters*)

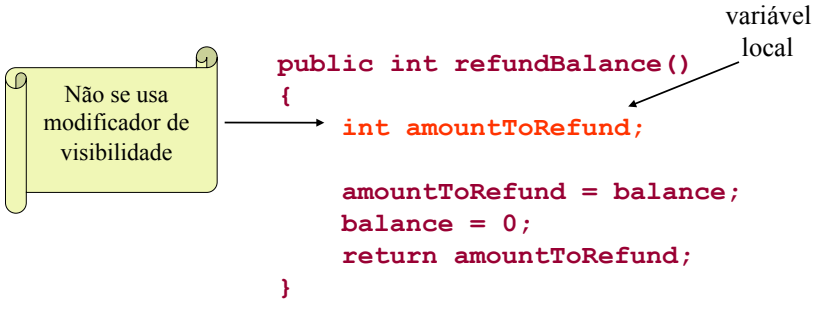
- Utilizados para modificar o estado de um objeto
  - Geralmente contêm instruções de atribuição
  - Geralmente recebem parâmetros
  - Geralmente seu tipo de retorno é void



Variáveis locais

- Atributos são um tipo de variável:
  - armazenam valores por toda a vida de um objeto
  - são acessíveis por meio da classe (seu escopo)
  - uma cópia do atributo por objeto instanciado
- Os métodos também podem declarar variáveis locais:
  - existem apenas enquanto o método está em execução
  - são acessíveis somente dentro do método

Variáveis locais



Resumo

- O corpo das classes pode conter atributos (campos), construtores e métodos
- Atributos armazenam o estado de um objeto
- Construtores inicializam objetos
- Métodos implementam o comportamento dos objetos
- Atributos, parâmetros e variáveis locais são variáveis
- Atributos persistem pelo tempo de vida de um objeto
- Parâmetros são utilizados para receber valores em um construtor ou método
- Variáveis locais são utilizadas para armazenamento temporário de curta duração e existem apenas durante a execução do método

Métodos estáticos

- Métodos estáticos (métodos de classe)
  - Aplicam-se à classe como um todo
  - Não é necessário criar um objeto para utilizá-los
  - São chamados especificando-se o nome da classe onde o método é declarado:

    ```
    NomeClasse.nomeMétodo()
    ```

  - Todos os métodos da classe Math são `static`:
    - Ex: `Math.sqrt(144.0)`

Atributos estáticos

- Campos estáticos (variáveis de classe)
  - A mesma cópia é compartilhada entre todas as instâncias da classe (objetos)
  - Uma espécie de "variável global" da classe
- Constantes
  - Declaradas com a palavra-chave **`final`**. Não pode ser alterada depois da inicialização
  - Classe Math:
    - Campos `Math.E` e `Math.PI` são `final static`

A classe Math

`abs( x )`	**valor absoluto de *x***	`abs( 23.7 )` é `23.7`
		`abs( 0.0 )` é `0.0`
		`abs( -23.7 )` é `23.7`

`ceil( x )`	**arredonda *x* para o menor inteiro não menor que *x***	`ceil( 9.2 )` é `10.0` `ceil( -9.8 )` é `-9.0`
`cos( x )`	**co-seno trigonométrico de *x* (*x* em radianos)**	`cos( 0.0 )` é `1.0`
`exp( x )`	**método exponencial e^x**	`exp( 1.0 )` é `2.71828` `exp( 2.0 )` é `7.38906`
`floor( x )`	**arredonda *x* para o maior inteiro não maior que *x***	`floor( 9.2 )` é `9.0` `floor( -9.8 )` é `-10.0`
`log( x )`	**logaritmo natural de *x* (base *e*)**	`log( Math.E )` é `1.0` `log( Math.E * Math.E )` é `2.0`
`max( x, y )`	**maior valor de *x* e *y***	`max( 2.3, 12.7 )` é `12.7` `max( -2.3, -12.7 )` é `-2.3`
`min( x, y )`	**menor valor de *x* e *y***	`min( 2.3, 12.7 )` é `2.3` `min( -2.3, -12.7 )` é `-12.7`
`pow( x, y )`	***x* elevado à potência de *y* (isto é, x^y)**	`pow( 2.0, 7.0 )` é `128.0` `pow( 9.0, 0.5 )` é `3.0`
`sin( x )`	**seno trigonométrico de *x* (*x* em radianos)**	`sin( 0.0 )` é `0.0`
`sqrt( x )`	**raiz quadrada de *x***	`sqrt( 900.0 )` é `30.0`
`tan( x )`	**tangente trigonométrica de *x* (*x* em radianos)**	`tan( 0.0 )` é `0.0`

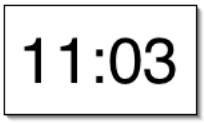
Cooperação entre objetos

Um relógio digital

Abstração e modularização

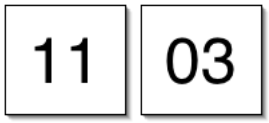
- **Abstração** é a habilidade de ignorar detalhes sobre as partes para concentrar a atenção no nível mais alto de um problema
- **Modularização** é o processo de dividir um todo em partes bem definidas, que podem ser construídas e examinadas separadamente e que interagem de uma maneira pré-determinada

Modularização no exemplo do relógio



Um mostrador de número de quatro dígitos?

Ou um mostrador de número de dois dígitos?



Implementação — NumberDisplay

```
public class NumberDisplay
{
    private int limit;
    private int value;

    Construtor e métodos omitidos
}
```

Uma classe de exibição de um dígito, com um limite superior e um valor atual

Implementação — ClockDisplay

```
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;

    Construtor e métodos omitidos
}
```

Um mostrador de relógio completo que contém, internamente, dois mostradores de números

Diagrama de objetos

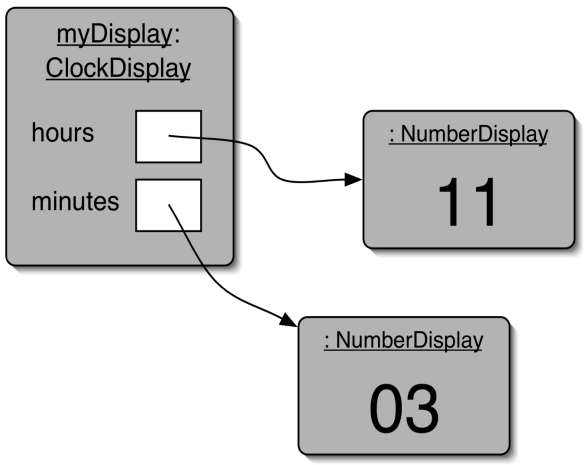
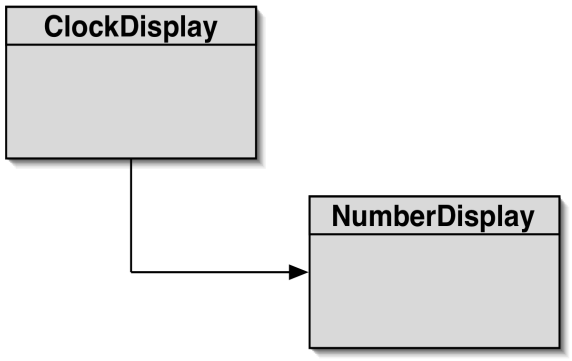
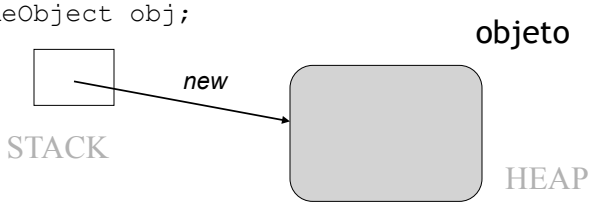


Diagrama de classes



Tipos primitivos vs. Objetos

```
SomeObject obj;
```



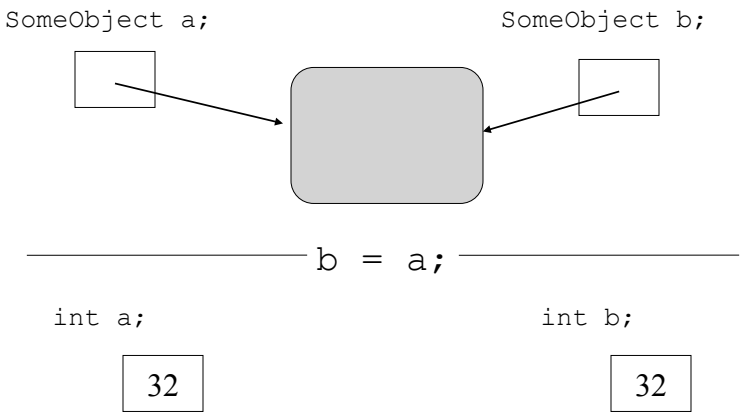
Uma referência para o objeto é armazenada na variável

```
int i;
```

tipo primitivo

Os tipos primitivos são armazenados na própria variável

Tipos primitivos vs. Objetos



NumberDisplay (1)

```
// Construtor
public NumberDisplay(int rollOverLimit)
{
    limit = rollOverLimit;
    value = 0;
}


public void increment()
{
    value = (value + 1) % limit;
}
```

NumberDisplay (2)

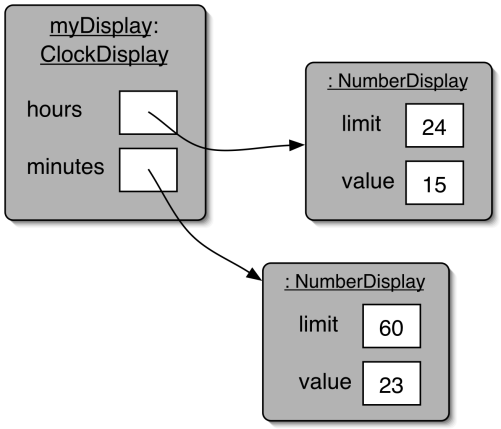
```
public String getDisplayValue()
{
    if(value < 10)
        return "0" + value;
    else
        return "" + value;
}
```


ClockDisplay: Objetos criando objetos

```
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;
    private String displayString;

    public ClockDisplay()
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        updateDisplay();
    }
}
```

Diagrama do objeto ClockDisplay



Objetos criando objetos

Na classe NumberDisplay:

```
public NumberDisplay(int rollOverLimit);
```

parâmetro formal

Na classe ClockDisplay:

```
hours = new NumberDisplay(24);
```

parâmetro real

ClockDisplay: Métodos externos

```
public void timeTick()
{
    minutes.increment();
    if(minutes.getValue() == 0) {
        // retornou a zero
        hours.increment();
    }
    updateDisplay();
}
```


ClockDisplay: Método interno

```
// Atualiza a string interna que
// representa o mostrador

private void updateDisplay()
{
    displayString =
        hours.getDisplayValue() + ":" +
        minutes.getDisplayValue();
}
```

ClockDisplay: Múltiplos construtores

```
public ClockDisplay() {
       hours = new NumberDisplay(24);
       minutes = new NumberDisplay(60);
       updateDisplay();  // Hora inicial 00:00
   }

public ClockDisplay(int hour, int minute) {
       hours = new NumberDisplay(24);
       minutes = new NumberDisplay(60);
       setTime(hour, minute);
   }
```

É possível inicializar um objeto ClockDisplay de duas maneiras diferentes –- *sobrecarga de construtor ou método*

A palavra-chave this

```
public class MailItem
{
    private String from;
    private String to;
    private String message;

    public MailItem(String from, String to,
                    String message)
    {
        this.from = from;
        this.to = to;
        this.message = message;
    }
}
```

A expressão *this* referencia o objeto atual, diferenciando o atributo do parâmetro

Relacionamento entre Objetos

Associação, Agregação e Composição

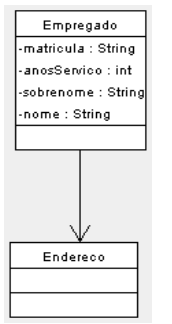
Tipos de relacionamentos

- Objetos **não** são ilhas isoladas. Pelo contrário, podem existir diferentes tipos de relacionamentos entre os objetos ao longo do ciclo de vida do sistema
- **Reuso de classe**: fornece **menos** flexibilidade
  - **Herança pura** (sobreposição): b "é" a
  - **Herança com extensão**: b "é um tipo de" a
- **Uso e reuso de objetos**: fornece **mais** flexibilidade
  - **Associação**: a "é usado por" b
  - **Agregação**: a "é parte de" b
  - **Composição**: a "é parte essencial de" b

O grau de coesão entre os objetos foi colocado do mais fraco para o mais forte

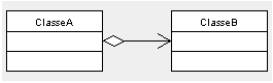
Associação

- Representa relacionamentos mais fortes entre instâncias de classes (objetos)

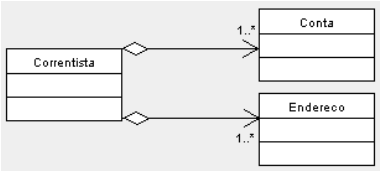



```
public class Empregado {
   private String matricula;
   private int anosServico;
   private String sobrenome;
   private String nome;
   private Endereco endereco;
  . . .
}
public class Endereco {
   . . .
}
```

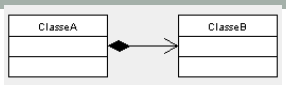

Agregação



- A agregação é um relacionamento do tipo todo/parte:
  - É usado para mostrar uma relação de inclusão lógica, ou seja, um todo formado por partes
  - Embora as partes possam existir independentemente do todo, sua existência é basicamente para formar o todo
  - Exemplo: um correntista **precisa** de pelo menos uma conta e um endereço. Sem uma conta não se pode dizer que ele é correntista
  - A agregação é frequentemente representada como uma simples associação

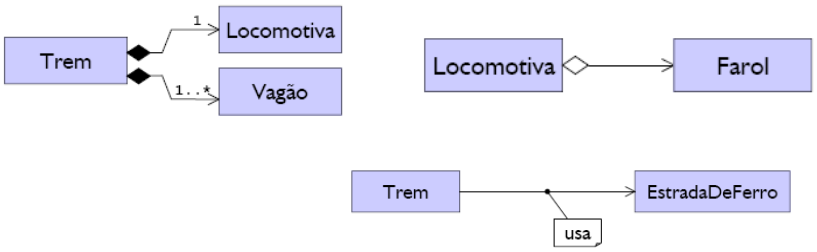


Composição



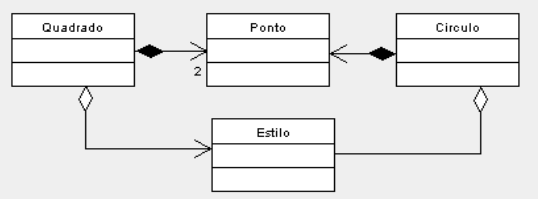
- Uma composição é um tipo de agregação. A diferença é que na composição o objeto composto faz parte de somente um relacionamento (um todo), enquanto que na agregação isso não é obrigatório
- Exemplo 1: um trem é formado por uma locomotiva e vagões. Uma locomotiva ainda tem um farol
- Exemplo 2: um Quadrado é formado por dois pontos e um Círculo tem um ponto e um raio. Cada uma dessas formas geométricas ainda tem um estilo

Composição X Agregação X Associação



- Um trem não existe sem a locomotiva e os vagões. Por sua vez, os dois últimos até "assumem" uma nova identidade ao formarem o trem
- Uma locomotiva possui um farol (mas não vai deixar de ser uma locomotiva se não o tiver)
- Um trem usa uma estrada de ferro (ela não faz parte do trem, mas ele depende dela)

Composição X Agregação

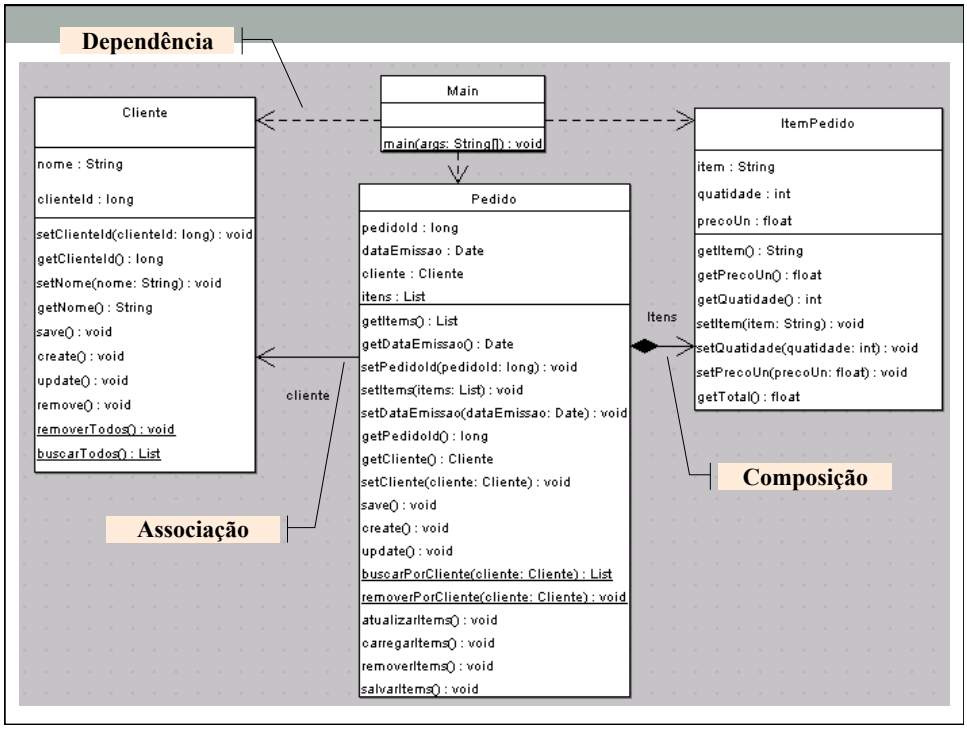


- A relação de **`Ponto`** com **`Circulo`** e **`Quadrado`** é uma composição, pois os mesmos não podem ser compartilhados
- Enquanto que o mesmo objeto de **`Estilo`** pode ser compartilhado por **`Circulo`** e **`Ponto`** (agregação)

Composição X Agregação

- Tradução de composição e agregação na linguagem Java

```
public class Quadrado {
   // p1 e p2 são composição - new
   // estilo é agregação - atribuição
   private Ponto p1, p2;
   private Estilo estilo;
   public Quadrado(int x1, int y1,
                   int x2, int y2, Estilo e){
         p1 = new Ponto(x1, y1);
         p2 = new Ponto (x2, y2);
         estilo = e;
   }
}
```



Generalização e Especialização

Herança e Polimorfismo

Herança

- Uma maneira de expressar herança é pelo termo **É-UM**
  - Fusca é um Carro
  - Gerente é um Funcionario

- Na linguagem Java, a palavra reservada **extends** representa este relacionamento É-UM:

```
public class Funcionario {
      . . .
}
```

```
public class Gerente extends Funcionario {
      . . .
}
```

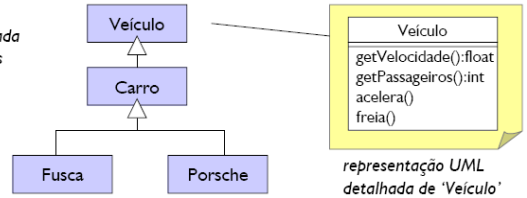

Herança (*reuso de interface*)

- Um carro é um veículo

representação UML

- Fuscas e Porsches são carros (e também veículos)

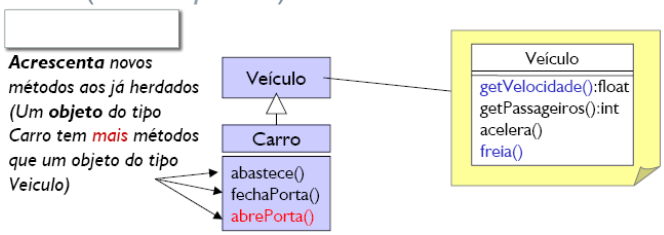
representação UML simplificada (não mostra os métodos)



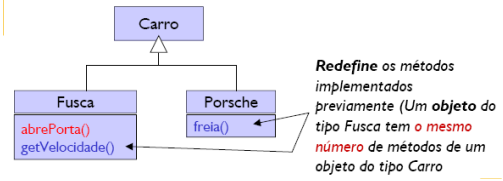
representação UML detalhada de 'Veículo'

Herança (*extensão e sobreposição*)

- Extensão (*é-um-tipo-de*)

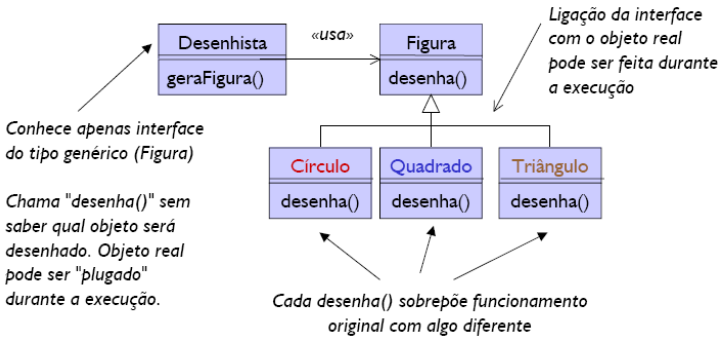


- Sobreposição (*é-um* - herança pura)



Polimorfismo

- Uso de um objeto no lugar de outro:
  - pode-se escrever código que não dependa da existência prévia de tipos específicos
  - mesmo nome do método, diferentes implementações



Resumo de características OO

- **Abstração de conceitos**
  - Classes: definem um tipo, separando interface de implementação
  - Objetos: instâncias utilizáveis de uma classe
- **Herança: "é um"**
  - Aproveitamento do código na formação de hierarquias de classes
  - Fixada na compilação (inflexível)
- **Associação "tem um"**
  - Representa relacionamentos entre objetos
  - Pode ter comportamento e estrutura alterados durante execução
  - Vários níveis de acoplamento: associação, composição, agregação
- **Encapsulamento**
  - Separação de interface e implementação que permite que usuários de objetos possam utilizá-los sem conhecer detalhes de seu código
- **Polimorfismo**
  - Permite que um objeto seja usado no lugar de outro (mesma interface, diferentes implementações)