Module II – Introduction to OO Programming

Prof. Ismael H F Santos

Considerações Gerais

- **Objetivo:** *Discutir os principais conceitos e os princípios básicos da Orientação a Objetos usando a linguagem C++.*
- **A quem se destina :** *Alunos e Profissionais que desejem aprofundar seus conhecimentos sobre Linguagem C++ e suas aplicações*

Bibliografia

- *Thinking in C++ 2nd Edition by Bruce Eckel.*
  - *http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html*
- *The Most Important C++ Books...Ever*
  - *http://www.artima.com/cppsource/top_cpp_books.html*
- *C++ Free Computer Books*
  - *http://www.computer-books.us/cpp.php*

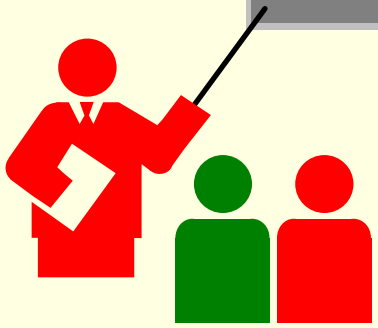
Webliografia

- CppNotes - Fred Swartz
  - http://www.tecgraf.puc-rio.br/~ismael/Cursos/apostilas/Cpp-Notes/index.html
- The Code Project - C++
  - http://www.codeproject.com/index.asp?cat=2
- C/C++ reference
  - http://www.cppreference.com/index.html
- Boost - free peer-reviewed portable C++ source libraries
  - http://www.boost.org/

Agenda

- *Programming Languages*
- *Paradigma OO*
- *Notacao UML*

Programming Languages



Introduction

- Definição de Linguagem de Programação
  - Uma Linguagem de Programação (LP) é uma notação especializada, a ser usada por uma ou mais pessoas, para expressar um processo (programa) através do qual um computador pode resolver um problema.
  - As LPs são linguagens definidas com uma estrutura restrita de forma a se poder especificar de forma precisa a execução de tarefas a serem realizadas pelo computador.

Introduction

- Características desejáveis de uma LP
  - **Confiabilidade**
    - A linguagem não deve induzir o programador a erros, em particular erros que não possam ser descobertos com facilidade.
  - **Manutenibilidade**
    - A linguagem deve ajudar o programador a corrigir erros, facilitando a sua identificação e a determinação da ação de correção, seja qual for a causa original do erro.
  - **Sintaxe e Semântica**
    - A LP deve ter regras claras que definam a sua Sintaxe (forma/estrutura) e a sua Semântica (significado), de forma que possamos julgar se uma determinada sentença é “bem formada”.

Introduction

- Características desejáveis de uma LP
  - **Eficiência**
    - Algoritmos devem poder ser implementados de forma eficiente quando codificados na LP. A estrutura da linguagem deve facilitar o processo de geração e otimização de código, permitindo a construção de compiladores que gerem código objeto que faça uso eficiente dos recursos de máquina disponíveis.
- Os objetivos de **Confiabilidade e Manutenibilidade** são favorecidos pelas seguintes qualidades de uma LP:

Introduction

- Qualidades de uma LP
  - ***Legibilidade** – É o critério mais importante. O programador deve ser capaz de ler e compreender programas sem encontrar ambigüidades.*
    - ***Simplicidade - a LP deve prover um número mínimo de conceitos e estruturas.***
    - ***Estruturas de Controle Claras e bem definidas***
    - ***Suporte a Abstração de Dados - criação de novos tipos de dados pelo usuário Tipos Abstratos de Dados - TAD. A nível de procedimentos a abstração facilita modularidade e boas práticas de projeto.***

Introduction

- Qualidades de uma LP
  - ***Ortogonalidade - refere-se a integração entre os conceitos, o grau de interação entre diferentes conceitos, e como eles podem ser combinados de maneira consistente.***

Simplicidade(-)	Ortogonalidade(-)	Ortogonalidade(+)

C/C++/Java:
 	
cont = cont+1;
cont += 1;
cont++;
++cont;
	IBM:
	
	A Reg,#MemAddr
	
	AR Reg1,Reg2
		VAX:
		
		ADDL Op1, Op2
		
		Opi, pode ser tanto Reg quanto #MemAddr

Introduction

- Critérios para avaliação de uma LP
  - ***Redigibilidade - Facilidade de escrita de um programas de maneira que seja natural para o problema. Linguagens de alto nível são mais Redigíveis que Linguagens de baixo nível.***
    - ***Fatores que influenciam: Simplicidade, Ortogonalidade e Suporte para Abstração (TADs).***

Introduction

- Critérios para avaliação de uma LP *:*
  - *Legibilidade x Redigibilidade em Pascal*

Pascal-1	Pascal-2	Pascal-3	Pascal-4

if x>1 **then**
  if x=2 **then**
  x:=3
else
  x:=4;
	if x>1 **then**
	  if x=2 **then**
	  x:=3
	  else
	  x:=4;
		if x>1 **then**
		  if x=2 **then**
		  x:=3;
		  else
		  x:=4;
			if x>1 **then**
			BEGIN
			  if x=2 **then**
			  x:=3;
			END
			else
			  x:=4;

Introduction

- Critérios para avaliação de uma LP
  - *Legibilidade x Redigibilidade em outras LPs*

C/C++	Algol	ADA	Fortran

if (x>1)
{
 if(x==2)
  x=3;
}
else
  x=4;
	if x>1 **then**
	  if x=2 **then**
	  x=3;
	  else
	  x=4;
	  fi
	fi
		if x>1 **then**
		  if x=2 **then**
		  x:=3;
		  else
		  x:=4;
		  end fi;
		end fi
			if (X.GT.1) **then**
			  if (X.EQ.2)
			then
			  x=3
			  else
			  x=4
			  endif
			endif

Introduction

- Critérios para avaliação de uma LP
  - *Manutenibilidade* **-** *facilidade de modificação dos programas para incorporar novos requisitos de projeto.*

C/C++

```
#define NPALS 65536

int mem[NPALS];


void AlocaMem( void )
{
 int adress;

 if( adress > NPALS )
  printf(
    "\nmemória out!");
}
```

PASCAL

```
CONST
 NPALS = 65536;
VAR   Mem: array
  [1..NPALS] of integer;

Procedure ALOCA_MEM;
VAR
 Adress: 1.. NPALS;
BEGIN
 If adress > NPALS then
  Writeln(
     'memoria out!');
END;
```

Fortran

```
PARAMETER (NPALS=65536)
INTEGER mem(NPALS)
COMMON /MEMORY/ mem


SUBROUTINE ALOCA_MEM
INTEGER mem(NPALS)
COMMON /MEMORY/ mem
INTEGER adress

IF(adress.GT.NPALS)THEN
  Print*,'memoria out!'
ENDIF
RETURN
END
```


Introduction

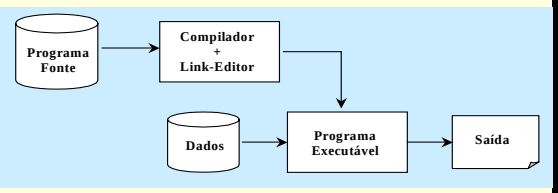
- Critérios para avaliação de uma LP
  - *Manutenibilidade - A linguagem deve ajudar o programador a corrigir erros facilitando a sua identificação;*
  - *Portabilidade - utilização de um padrão independente de máquina, permitindo a movimentação do programa entre plataformas de forma "fácil".*
  - *Eficiência - A estrutura da linguagem deve facilitar o processo de geração e otimização de código.*

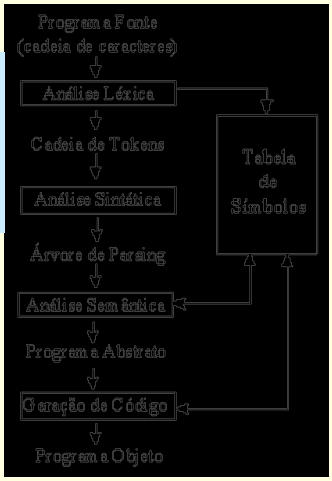
Introduction

- Critérios para avaliação de uma LP
  - *Confiabilidade - A linguagem não deve induzir o programador a erros, em particular erros que não possam ser descobertos com facilidade*.
    - *Fatores que influenciam: Checagem de Tipos, Tratamento de Exceções, Ausência de Aliasing.*
  - Custo:
    - Treinamento dos Programadores;
    - Criação de Software - Empacotamento;
    - Compilação e Custo do Compilador;
    - Execução x Interpretação
    - Depuração, Ambientes de Desenvolvimento e Manutenção;

Compiling x Interpretation

- Compilação + Link-Edição



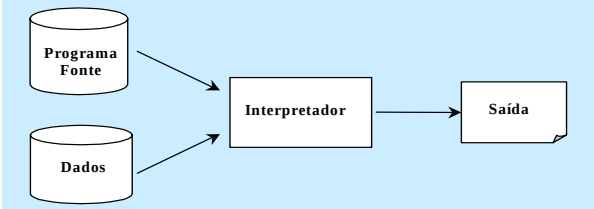


- As fases principais da Compilação

Compiling x Interpretation

- Interpretador
  - busca na próxima instrução a ser executada;
  - análise da instrução, determinando a ação que deve ser executada, como devem sei obtidos os dados de entrada para a execução dessa ação, e a forma de tratamento dos seus resultados;
  - busca dos dados necessários;
  - execução da ação correspondente sobre esses dados
  - armazenamento, (ou outro tratamento adequado) dos resultados

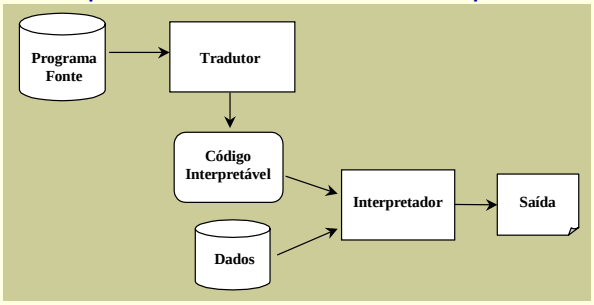
Linguagens interpretadas servem para a realização de uma prototipagem rápida.

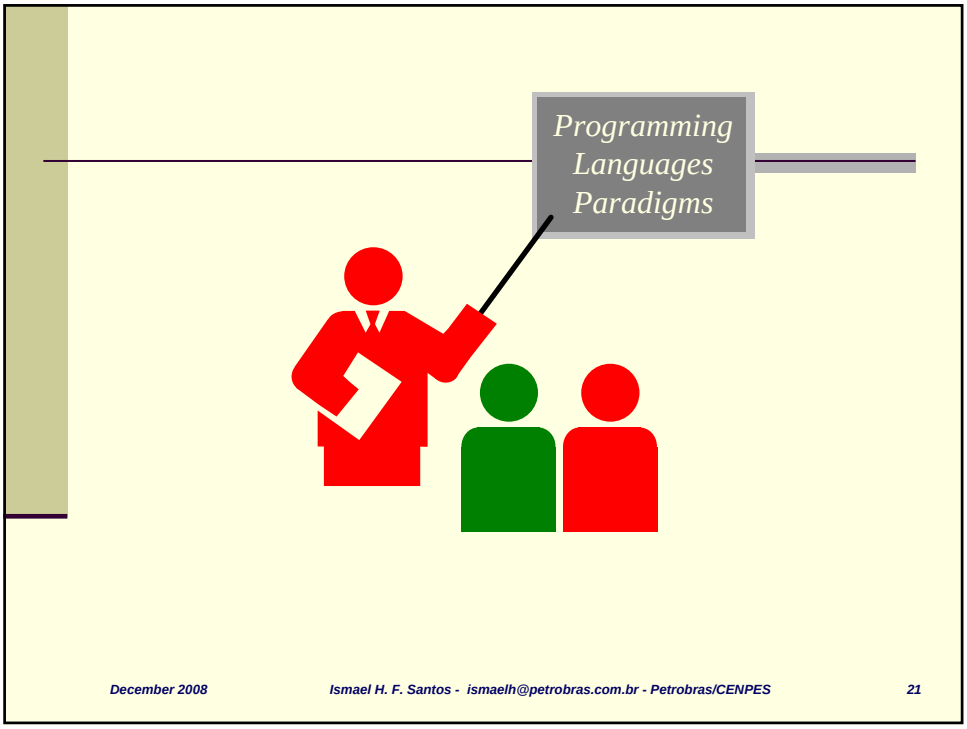


Compiling x Interpretation

- Alguns **Interpretadores** efetuam urna tradução do código fonte para uma representação interna ou código intermediário, cuja interpretação pode então ser feita com maior facilidade.
- Às vezes o programa responsável pela tradução acima mencionada é de Interpretador, às vezes de Compilador

Este é o caso das plataformas Java e .NET





Programming Languages Paradigms

- Programação Procedural ou Imperativo
- Programação Funcional
- Programação Lógico
- Programação Orientada por Objetos

Paradigma Procedural

- Foco é no processamento, isto é, no algoritmo necessário para se fazer a computação desejada.
- Técnica: *Decida que procedimentos são necessários; use os melhores algoritmos encontrados para eles*.
- Este estilo de programação discute, entre outros assuntos, os modos de passagem de parâmetros, maneiras de distinguir diferentes tipos de parâmetros, diferentes tipos de funções (procedimentos, rotinas, macros,...). Fortran, C e Pascal são linguagens de programação que dão suporte a esta técnica.

Paradigma Procedural

- Linguagens Imperativas ou Procedurais
  - caracterizam-se por um processamento basicamente seqüencial e uso intenso da atribuição. As variáveis descrevem o estado da computação a cada instante. São linguagens apropriadas para máquinas cuja arquitetura é baseada na máquina de Von Newmann. Podemos dividir as Linguagens imperativas em dois tipos:
    - **Estrutura Procedural Simples - Fortran, Cobol, Algol, PL/I, C;**
    - **Estrutura Procedural em Blocos - Pascal**

Paradigma Funcional

- Linguagens Aplicativas ou Funcionais
  - neste paradigma em vez de nos concentrarmos na seqüência de transições de estados que a computação deve realizar até obtermos uma resposta, nós nos concentramos na seqüência de funções que que devem ser aplicadas aos dados para que a partir do estado inicial nos obtenhamos uma resposta. Exemplos de linguagens deste paradigma são o **Lisp** e o **ML**;

Paradigma Funcional - Example

Programa HelloWord

```
$ lisp -> Para carregar o interpretador lisp
// Predicado que imprime a mensagem Hello World na tela.
(define (hello i)
       (cond ((eq i 2) (print "Hello World"))
             (T (print "Goodbye World"))
       )
)
// No prompt do interpretador obteríamos:
>(hello 2)
> Hello World
```

Paradigma Lógico

- Linguagens Baseadas em Regras ou Lógicas
  - a computação é realizada por um motor de inferência que a partir de um conjunto de regras fornecido (programa), e uma pergunta feita pelo usuário tenta-se verificar a veracidade ou não da pergunta. **Prolog (PROgraming in LOGic)** é a linguagem mais famosa deste paradigma. Outros exemplos são **YACC(Yet Another Compiler Compiler), Lex, SNOBOL.**

Paradigma Lógico - Example 1

Programa HelloWord

```
$ prolog -> Para carregar o interpretador
// Predicado que imprime a mensagem Hello World na tela.
writeit :- write('Hello World'), nl.

// No prompt do interpretador obteríamos:
?- writeit.
Hello World
Yes.
```

Paradigma Lógico - Example 2

Programa Verificação de Data

// O programa abaixo retornará Yes a qualquer pergunta sobre a validade de uma data dada.

?- date(1964, 12, 31);

Yes.

?- date(1999, 2, 29);

No.

// Regras de conhecimento previamente carregadas pelo interpretador, através do comando abaixo:

?- load('date.pgm.prolog');

Example 2 (cont'd)

```
mdays(_, 1, 31).
mdays(Y, 2, 28):- not leap(Y).
mdays(Y, 2, 29):- leap(Y).
mdays(_, 3, 31).
mdays(_, 4, 30).
mdays(_, 5, 31).
mdays(_, 6, 30).
mdays(_, 7, 31).
mdays(_, 8, 31).
mdays(_, 9, 30).
mdays(_, 10, 31).
mdays(_, 11, 30).
mdays(_, 12, 31).
```

```
leap(Y) :- divides(4, Y),
           not divides(100, Y).
leap(Y) :- divides(400, Y).

divides(Div, K) :- 0 is K mod Div.
```

Paradigma Objetos

- Linguagens Orientadas a Objeto
  - neste paradigma objetos complexos são construídos, e um conjunto limitado de operações sobre estes objetos são definidas. Novos objetos mais complexos podem ser construídos a partir dos anteriores. De uma certa forma neste paradigma nós estamos tentando fundir o melhor do paradigma Imperativo com o melhor do paradigma Funcional. Exemplos de linguagens que implementam este paradigma são o **Simula 67, Ada, Smalltalk, C++, Java, Eiffel etc;**

Paradigma Objetos x Programação Modular

- Ao longo dos anos, a ênfase no projeto de programas tem se transferido do projeto de procedimentos para a organização dos dados
- Entre outras coisas, isto reflete um aumento no tamanho dos programas. Um conjunto de procedimentos relacionados com os dados que eles manipulam é freqüentemente chamado de *módulo*.
- Técnica: *Decida que módulos você quer; particione o programa de maneira que os dados fiquem escondidos (protegidos) nos módulos.*

Paradigma Objetos x Programação Modular (cont'd)

- Onde não existe um agrupamento de funções relacionadas com um determinado dado, a programação procedural é usada. As técnicas para projetar "bons procedimentos" são agora aplicadas a cada função em um módulo. Modula-2 dá suporte a este estilo de programação, enquanto C meramente permite o seu uso.
- Apesar de C++ não ter sido especificamente projetado para suportar programação modular, seu conceito de classe prove um suporte ao conceito de módulo. Além disso, C++ suporta a noção de módulos de C, que são implementados através de unidades de compilação separadas (extern "C").

Data Abstraction

- A programação com módulos permite centralizar todo o controle de um determinado tipo de dado em um módulo gerente deste tipo.
- Porém, "tipos" implementados deste modo são claramente diferentes dos tipos pré-definidos da linguagem. Cada módulo gerente de um tipo deve definir um mecanismo separado para criação de variáveis de seu tipo, não existindo nenhuma norma para atribuição de identificadores de instâncias do tipo.

Data Abstraction

- Em outras palavras, o conceito de módulos, que suporta o paradigma de *data hiding*, simplesmente permite o uso da técnica de abstração de dados mas não lhe dá um suporte adequado.
- Linguagens como Ada, Clu e C++ resolvem este problema permitindo ao usuário definir tipos que se comportam de uma maneira muito semelhante a dos tipos pré-definidos. Esses tipos são normalmente chamados de TAD ou *tipo abstrato de dado*.
- Onde não existe a necessidade de mais de um objeto de um tipo, a programação com módulos é suficiente.
- Técnica: Decida que tipos são necessários; ofereça um conjunto completo de operações para cada tipo.

OO Programming

- Um tipo abstrato de dados (*TAD*) define uma espécie de caixa preta. Uma vez que ele tenha sido definido, ele não tem como interagir com o resto do programa.
- Não existe um modo de adaptá-lo para novos usos, exceto pela modificação de sua definição. Isto pode acarretar em várias inflexibilidades e inconveniências, tal como a introdução de erros em um código que já era confiável mas teve que ser alterado para permitir a extensão de um TAD.
- A possibilidade de se estender um determinado TAD, ou melhor, especializar um TAD através da criação de um outro, é a característica mais marcante do paradigma de programação orientada a objetos.

OO Programming

- Linguagens que ofereçam algum mecanismo de extensão de TAD, como o mecanismo de herança introduzido por Simula e adotado por C++, são consideradas linguagens OO. De um modo geral, na literatura de programação e linguagens OO, os TADs recebem o nome de *classes* e suas instâncias são chamadas de objetos.

- Técnica: Decida que classes são necessárias; ofereça um conjunto completo de operações para cada tipo; torne explícita a similaridade usando herança.

OO Programming

- Uma das dificuldades iniciais da programação OO é a identificação das classes pertinentes a um sistema.

- Programas são usados para se obter respostas para certas questões do ambiente externo ao sistema (como em programas feitos para resolver um problema), para interagir com o mundo (como em um sistema de controle de processos), ou para criar novas entidades (como em um editor de texto ou um compilador).

OO Programming

- Um sistema bem projetado pode ser visto como um modelo operacional de aspectos de entidades reais.
- O mundo a ser modelado é composto de objetos, e seria apropriado organizar o modelo através de representações lógicas desses objetos. Os objetos de um sistema simplesmente refletem objetos externos.
- Definição: *a técnica de programação orientada a objetos considera a construção de sistemas de software como uma coleção estruturada de implementações de TADs tipos abstratos de dados. Estes conceitos estão de acordo com o princípio de information hiding.*

OO Programming

- Se uma classe é a implementação de um TAD, sua interface corresponde exatamente aos serviços da especificação do TAD.
- Por exemplo, a interface de uma classe pilha, oferecendo uma visão coerente de pilhas para o ambiente externo, incluiria serviços como push, pop, top e empty.
- A classe pode conter outras operações auxiliares usadas internamente para propósitos de implementação, mas estas não devem fazer parte da interface do TAD.

OO Programming

- Classes devem ser projetadas para serem reutilizadas em sistemas diferentes. Assim, o desenvolvimento de sistemas é visto como uma construção *bottom-up* a partir das classes existentes.
- Existem dois tipos de relacionamentos entre classes : cliente (agregação) e descendente (herança).
  - Agregação - Uma classe faz uso dos serviços da outra classe, definidos em sua interface. Por exemplo, uma classe de pilha poderia usar um *array* para sua implementação, e assim ser um cliente de uma classe array.
  - Herança - Uma classe é descendente de uma ou mais classes quando ela é projetada como uma extensão ou especialização dessas classes.

OO Programming Languages

- Principais características desejáveis:
  - *Estrutura modular baseada em objetos:* sistemas são modularizados com base em suas estruturas de dados.
  - *Gerenciamento automático de memória:* objetos não mais usados devem ser liberados pelo sistema de suporte da linguagem, sem a intervenção do programador.
  - *Abstração de dados:* objetos devem ser descritos como implementações de tipos abstratos de dados.

OO Programming Languages

- Principais características desejáveis
  - *Classes*: todo tipo não-primitivo é um módulo (classe), e todo módulo de alto nível é um tipo
  - *Herança:* uma classe pode ser definida como uma extensão ou restrição de outras.
  - *Polimorfismo e Ligação Tardia (Late Binding):* entidades do programa devem poder fazer referência a objetos de mais de uma classe, e as operações devem poder ter diferentes implementações em cada classe, que em tempo de execução são selecionadas automaticamente.

Example: TAD Pilha

- Um TAD Modela uma estrutura de dados através de sua funcionalidade. Define a interface de acesso à estrutura não fazendo qualquer consideração com relação à implementação.
- Pilha -> Funcionalidade: armazenagem LIFO
- Pilha -> Interface:

```
int isEmpty()
  verifica se a pilha está vazia
void push(int n)
  empilha o número fornecido
int pop()
   desempilha o número do topo e o retorna
int top()
  retorna o número do topo
```

Example: TAD Pilha

- A implementação de um TAD Pilha pode ser realizada por meio de uma classe. A classe deve prover todos os métodos definidos na interface do TAD. Um objeto dessa classe implementa uma instância do TAD

```
struct Stack {  // struct Stack -> public default access
    int topIndex_;                                  // <- Membros
    int[] data_;
    Stack(int size = 5) {                           // <- Construtor
       data = new int[size];  topIndex_ = -1;
     }
    int isEmpty() { return (topIndex _< 0) ? 1 : 0; }  // <- Métodos
    void push(int n) { data[++ topIndex_] = n; }
    int pop() { return data[topIndex_--]; }
    int top() { return data[topIndex_]; }
}
```


Fundamentos da Linguagem

- Uso da classe Pilha declarada

```
#include <iostream>
.....
Stack s(2);                 // Pilha para 2 números.
s.push(10);
s.push(20);
s.push(s.pop()+s.pop());            // s <- 20 + 10
std::cout<< s.top();                // 30
std::cout<< s.isEmpty();            // 0 -> false !
std::cout<< s.pop();                // 30 out
std::cout<< s.isEmpty();            // 1 -> true !
```

Fundamentos da Linguagem

- Encapsulamento
  - Na classe Stack implementada, nós encapsulamos a definição de pilha que desejávamos, porém, por falta de controle de acesso, é possível forçar situações nas quais a pilha não se comporta como desejado.

```
#include <iostream>
.....
Stack s(10);
std::cout<< s.isEmpty();  // 1 -> true!
s.push(6);
std::cout<< s.isEmpty();  // 0 -> false !

s.topIndex_ = -1;
std::cout<< s.isEmpty();  // 1 -> true!
```


Fundamentos da Linguagem

- Encapsulamento - Controle de Acesso
  - As linguagens OO disponibilizam formas de controlar o acesso aos membros de uma classe. No mínimo, devemos poder fazer diferença entre o que é público e o que é privado.
  - Membros públicos podem ser acessados indiscriminadamente, enquanto os privados só podem ser acessados pela própria classe.

Example: TAD Pilha – data hidding

```
class StackVE {  // pilha implementa com vetor
 private:
    int topIndex_;                // <- membros
    int[] data_;
 public:
    StackVE(int size) {           // <- Construtor
      data_ = new int[size]; topIndex_= -1;
    }
    int isEmpty() { return (topIndex_< 0) ? 1 : 0; }
    void push(int n) { data_[++topIndex_] = n; }
    int pop() { return data_[topIndex _--]; }
    int top() { return data_[topIndex_]; }
 }
// Exercício: Implemente o método isFull()
```


Example: TAD Pilha – data hidding

- Exemplo de Controle de Acesso
  - Com a nova implementação da pilha, o exemplo anterior não pode mais ser feito pois teremos um erro de compilação.

```
#include <iostream>
.....
StackVE s = new StackVE(10);
s.push(6);
s.top_index = -1;    // ERRO! na compilação
std::cout<< s.isEmpty();
```

Example: TAD Pilha – data hidding

```
class StackLE {  // pilha implementa com lista encadeada
private:
   struct  elemPilha {
      elemPilha *next_; int value_;
      elemPilha(elemPilha *p, int v) { next_=p; value_=v; }
   };
   elemPilha* top_;
public:
   // Constructor & Destructor
   StackLE();
   ~StackLE(void);
    // Class methods
   void push(int);
   int  pop();
   int  isEmpty();  // Exercício: Implemente todos os métodos
   int  isFull();
   int  top();
};
```

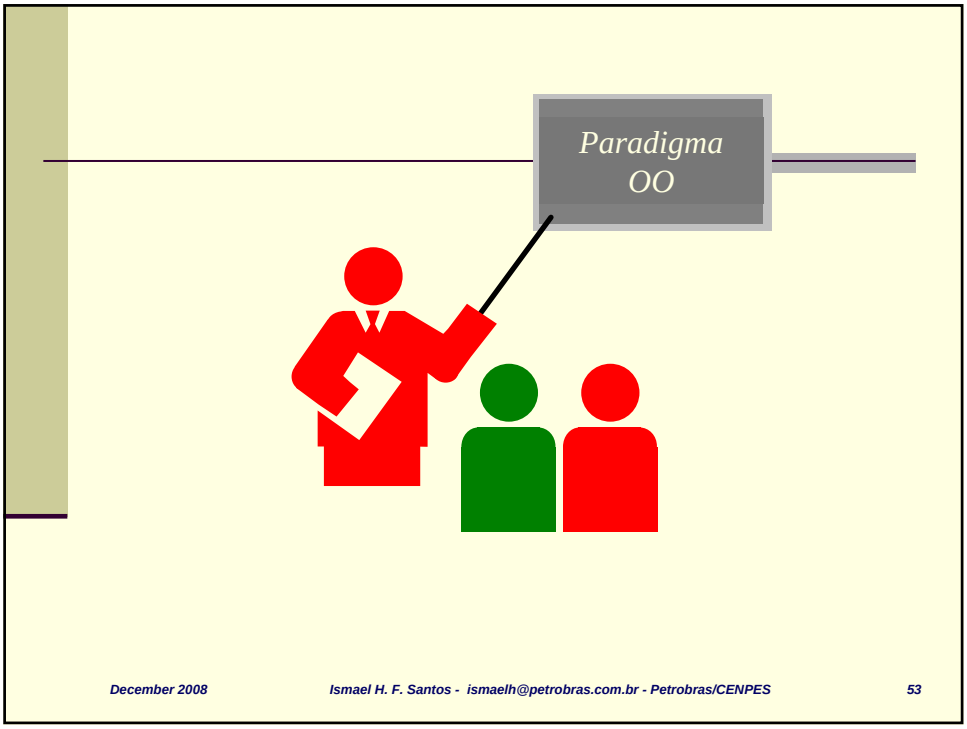

Advanced Exercise !

stack.h

```
//Interface Stack - classe com
  todos metodos abstratos
class Stack {
public:
  virtual void push(int) = 0; //
  metodos abstrato ...
  virtual int  pop()     = 0; //
  Late binding !!!
```

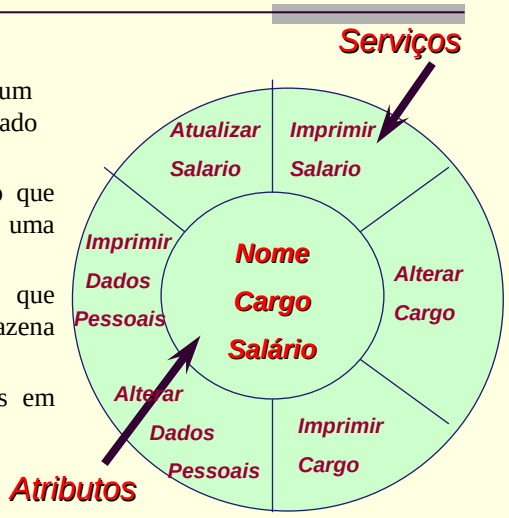


```
  virtual int  isEmpty() = 0;
```



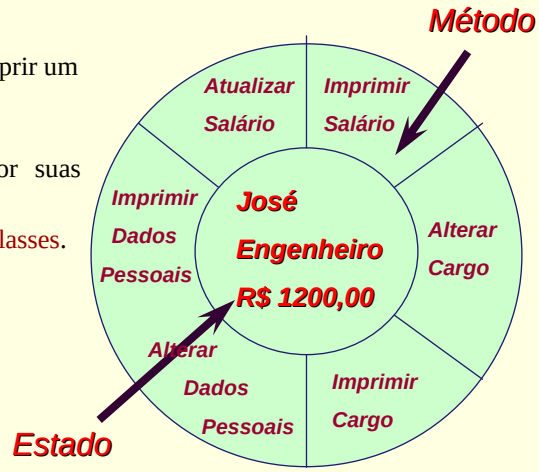
Paradigma OO

- Classes
  - O conjunto de requisições que um objeto pode cumprir é determinado pela sua classe.
  - Na classe defini-se o método que será executado para cumprir uma requisição.
  - A classe especifica que informações um objeto armazena internamente.
  - Classes podem ser compostas em hierarquias, através de herança.



Paradigma OO

- Objetos
  - Estão preparados para cumprir um determinado conjunto de requisições.
  - Estado é representado por suas informações internas.
  - Objetos são instâncias de classes.



Classes in C++

- Classes de C++ são extensões das struct de C

Exemplo em C

```
struct date { int month, day, year; };
typedef struct date Date;
Date today;
void setDate(Date * d, int month, int day, int year) {
   d->month = month;
   d->day   = day;
   d->year  = year;
}
void nextDate(Date * d){
   ...
}
void printDate(const Date* d) {
   ...
}
```

Classes in C++

- Em C, não existe nenhuma conexão explícita entre as funções de manipulação e o tipo de dado. Esse problema é resolvido por C++ através do uso de funções membro para manipular a estrutura de dados.

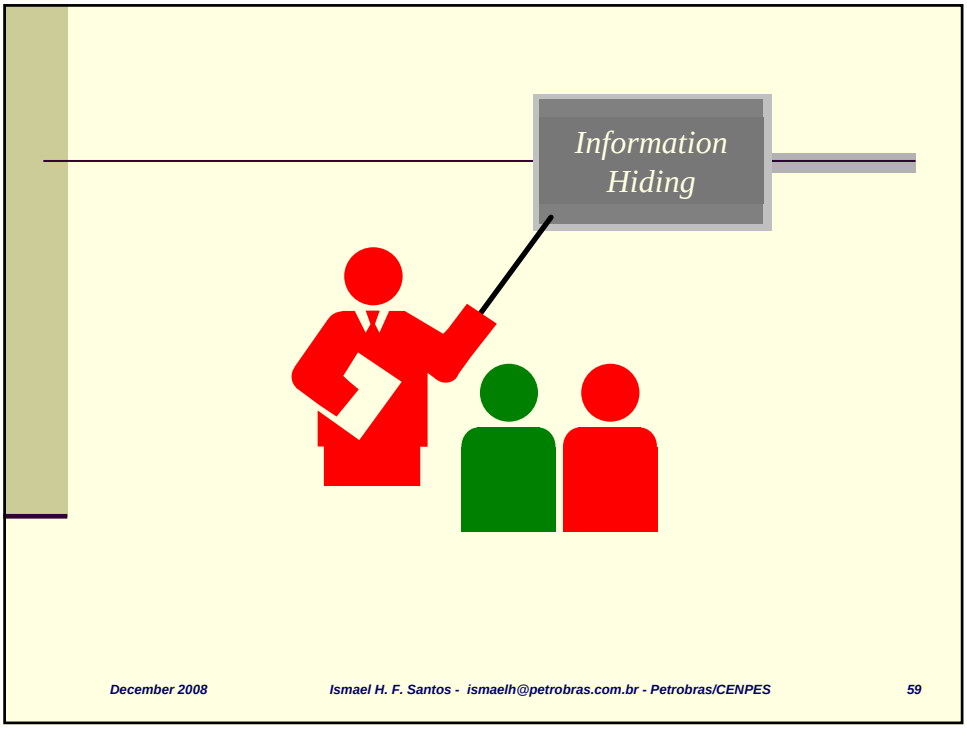
Exemplo em C++

```
struct Date {
   int month, day, year;

   void set(int,int,int);
   void get(int*,int*,int*);
   void next();
   void print();
};
void Date::set(int day, int month, int year){
   this->day   = day;     // Date *const this,
   this->month = month;   // this = &var -> erro !!!
    this->year  = year;    // O que aconteceria se fosse:
 }                         // const Date *const this
```


Classes in C++

- Métodos são declarados desta forma e só podem ser chamados sobre uma variável específica do tipo apropriado, usando a sintaxe padrão para acesso de membros de uma estrutura:.

```
Date today;
Date myBirthday;

void f() {
   myBirthday.set(15,8,1970);
   today.set(15,5,1995);

   myBirthday.print();
    today.next();
   today.print();
}
```



Information Hiding

- Um TAD define somente a sua interface, ou seja, como ele é manipulado. A implementacao é uma caracterítica particular que não é explicitada.
- Encapsulamento - Os atributos do TAD, ou seja, os detalhes dependentes da implementação, não precisam e não devem estar disponíveis para o usuário de um objeto, pois este deve ser acessado exclusivamente através da interface definida.
- Em C++ as classes representam os TADs e possuem uma parte invisível, que é a sua implementação, e uma parte visível que é a sua interface. As operações da interface possibilitam o acesso aos objetos.

Information Hiding Mechanism

- Para permitir a ocultação da informação do TAD, é necessário impor restrições na maneira em que uma classe pode ser acessada, isto é, que dados e quais métodos código podem ou não ser usados externamente.
- Existem três tipos de usuário de uma classe:
  - A própria classe
  - Usuários externos
  - Classes derivadas (este ponto será visto posteriormente)
- Cada um destes três tipos tem privilégios de acesso diferenciados. Cada um destes níveis tem uma palavra reservada associada:
  - **private**
  - **public**
  - **protected**

Information Hiding Mechanism

- Métodos são declarados usando o mecanismo de visibilidade (public, private ou protected) e só podem ser invocados sobre objetos do tipo apropriado, usando a sintaxe padrão para acesso de membros de uma estrutura.
- Os exemplos abaixo são equivalentes e ilustram o uso destas novas palavras reservadas

```
struct Sample {
 private:
   int a;
   int privateFunction(char* b);
 protected:
   int b;
   int protectedFunction(float);
 public:
   int c;
   float d;
   void publicFunction(controle*);
};
```

```
struct Sample {
 private: int a;
 protected: int b;
 public:
   int c;
   float d;
   void publicFunction(controle*);
 private:
  int privateFunction(char* b);
 protected:
   int protectedFunction(float);
};
```

Information Hiding Mechanism

- A diferença entre declarações com **struct** e **class** está no nível de proteção *default*. Caso não seja especificado nenhum qualificador, os membros de uma **struct** são públicos, enquanto que em uma **classe**, os membros são privados.

```
class Sample {
   int value;
   void f1(){ value = 0; }
   int  f2() { return value; }
};

int main() {
   Sample p;    // cria um objeto do tipo Sample
                // todos os membros são private,
               // não se pode fazer nada com p

   p.f1();                    // inválido
   printf("%ld", p.f2());     // inválido

   return 0;
}
```


Inheritance

Inheritance

- Herança é o recurso que torna o conceito de classe mais poderoso. Em C++, o termo herança se aplica apenas às classes. Variáveis não podem herdar de outras variáveis e funções não podem herdar de outras funções.

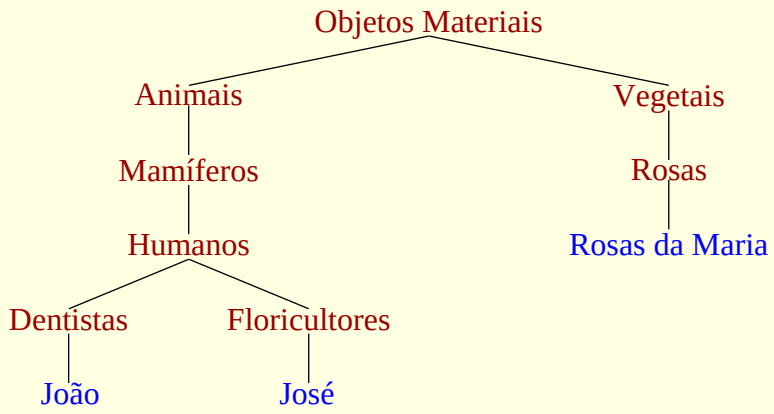
- A definição de classes em termos de outras classes constitui uma hierarquia:
  - **superclasses (classes ancestrais ou bases)**
  - **subclasses (classes derivadas)**

- Recurso utilizado para especializar ou estender classes;

Inheritance

- Especialização × Extensão
  - Uma classe pode herdar de outra para especializá-la redefinindo métodos, sem ampliar sua interface.
  - Uma classe pode herdar de outra para estendê-la declarando novos métodos e, dessa forma, ampliando sua interface.
  - Ou as duas coisas podem acontecer simultaneamente.
  - Herança implica tanto herança de interface quanto herança de código.

Simple Inheritance



Simple Inheritance

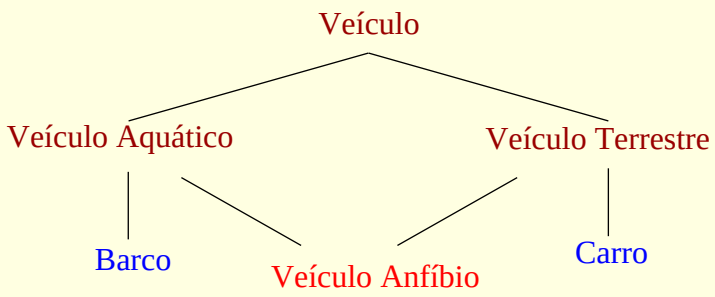
- Em C++, a classe Box é chamada classe base para a classe ColorBox, que é chamada classe derivada. A classe ColorBox foi declarada com apenas uma função, mas ela herda duas funções e duas variáveis.

```
class Box {
 public:
   int height, width;
   void Height (int h) { height=h; }
   void Width (int w) { width=w; }
};
class ColorBox : public Box {  // herança para extensão
  public:
   int color;
   void Color(int c) { color=c; }
};
int main() {
   ColorBox cb;
   cb.Color(5);
   cb.Height(3);  // herdada
   cb.Width(50);  // herdada
}
```

Interface x Code Inheritance

- Herança de interface significa que a classe que herda recebe todos os métodos declarados pela superclasse que não sejam *privados*.
- Herança de código significa que as *implementações* desses métodos também são herdadas. Além disso, os campos que não sejam privados também são herdados.

Multiple Inheritance



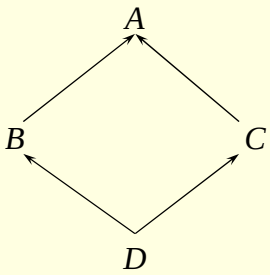
Simple x Multiple Inheritance

- O tipo de herança que usamos até agora é chamado de herança simples pois cada classe herda de apenas uma outra. Existe também a chamada herança múltipla onde uma classe pode herdar de várias classes.

- Herança múltipla não é suportada por todas as linguagens OO, pois apresenta um problema quando construímos hierarquias de classes onde uma classe herda duas ou mais vezes de uma mesma superclasse. O que, na prática, torna-se um caso comum.

Multiple Inheritance (cont)

- Problemas de Herança Múltipla
  - O problema de herdar duas vezes de uma mesma classe vem do fato de existir uma herança de código.
  - Inúmeras vezes, quando projetamos uma hierarquia de classes usando herança múltipla, estamos, na verdade, querendo declarar que a classe é *compatível* com as classes herdadas. Em muitos casos, a herança de código não é utilizada.

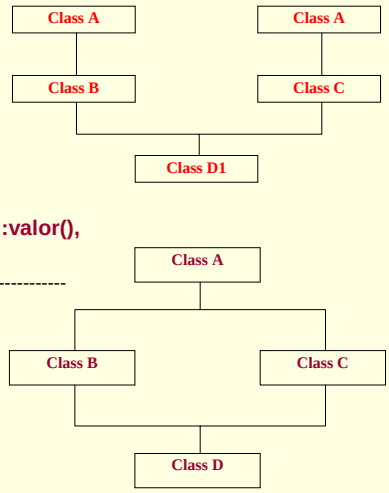


Multiple Inheritance in C++

```
struct A { int a; };
class B : public A {};
class C : public A {};

class D1 : public B, public C {
 public:
   int valor(){ return a; }
};
```

"Field 'a' is ambiguous in 'D1' in function D1::valor(), Could be C::a or B::a" !!!!

```
class D2 : public virtual B,
           public virtual C {
 public:
   int valor() { return a; }
};
```



Multiple Inheritance em C++

- *Em linguagens como C++, uma classe pode herdar métodos de duas ou mais classes*
  - *A classe resultante pode ser usada no lugar das suas duas superclasses via upcasting*
  - *Vantagem de herança múltipla: mais flexibilidade*
- *Problema*
  - *Se duas classes A e B estenderem uma mesma classe Z e herdarem um método x() e, uma classe C herdar de A e de B, qual será a implementação de x() que C deve usar? A de A ou de B?*
  - *Desvantagem de herança múltipla: ambigüidade. Requer código mais complexo para evitar problemas desse tipo*

Multiple Inheritance in C++

- Existem casos onde desejamos de fato ter multiplas cópias de uma classe base na classe derivada, vejamos:

```
class Scrollbar {
private: int x; int y;
public:  void Scrollbar(units n);
  //...
};
class HorizontalScrollbar : public Scrollbar {/*..*/};
class VerticalScrollbar   : public Scrollbar {/*..*/};
```

- MultiScrollwindow com vertical e horizontal scrollbars:

```
class MultiScrollWindow: public VerticalScrollbar,
                         public HorizontalScrollbar {/*
  ..*/};
MultiScrollWindow msw;
msw.HorizontalScrollbar::Scroll(5); // scroll left
msw.VerticalScrollbar::Scroll(12);  //...and up
```

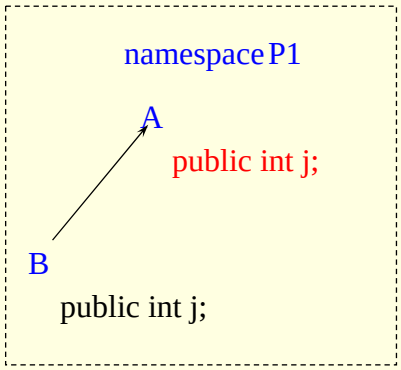
- Neste caso dois distintos sub-objetos Scrollbar devem existir (up and down e left and right).

Information Hiding

- Visibilidade & Herança
  - Membros **públicos** são herdados, enquanto membros privados não são. Às vezes precisamos algo intermediário: um membro que não seja visto fora da classe mas que possa ser herdado. As linguagens OO tipicamente dão suporte a esse tipo de acesso.
  - C++ permite declararmos um membro que, embora não seja acessível por outras classes, é herdado por suas sub-classes. Para isso usamos o modificador de controle de acesso protected.
- Resumo de Visibilidade
  - private: membros que são vistos só pelo própria classe e não são herdados por nenhuma outra;
  - protected: membros que são vistos pela classe e herdados por qualquer outra classe derivada como privados;
  - public: membros são vistos e herdados por qualquer classe.

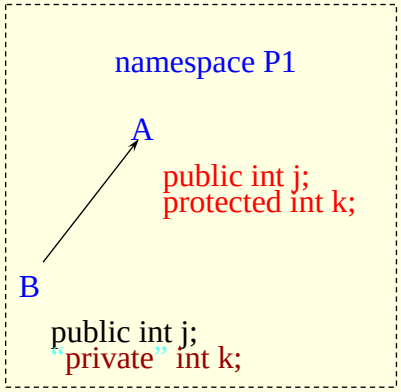
Information Hiding

- Herança de Membros



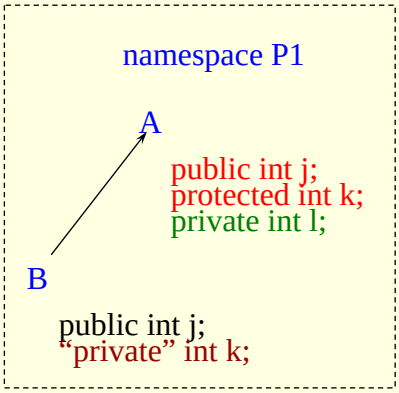
Information Hiding

- Herança de Membros



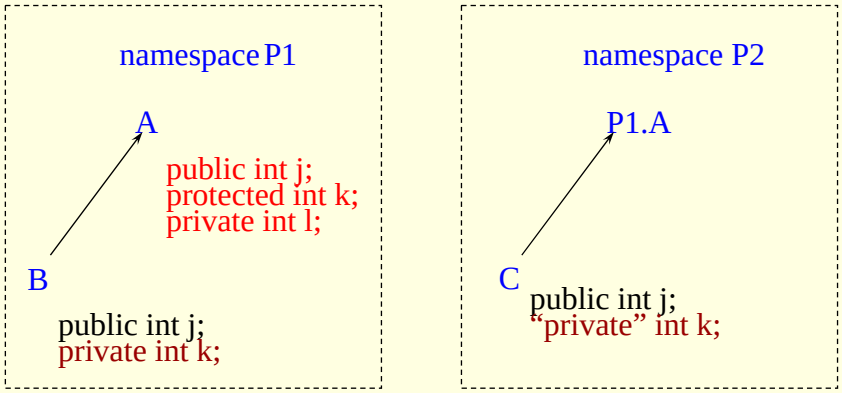
Information Hiding

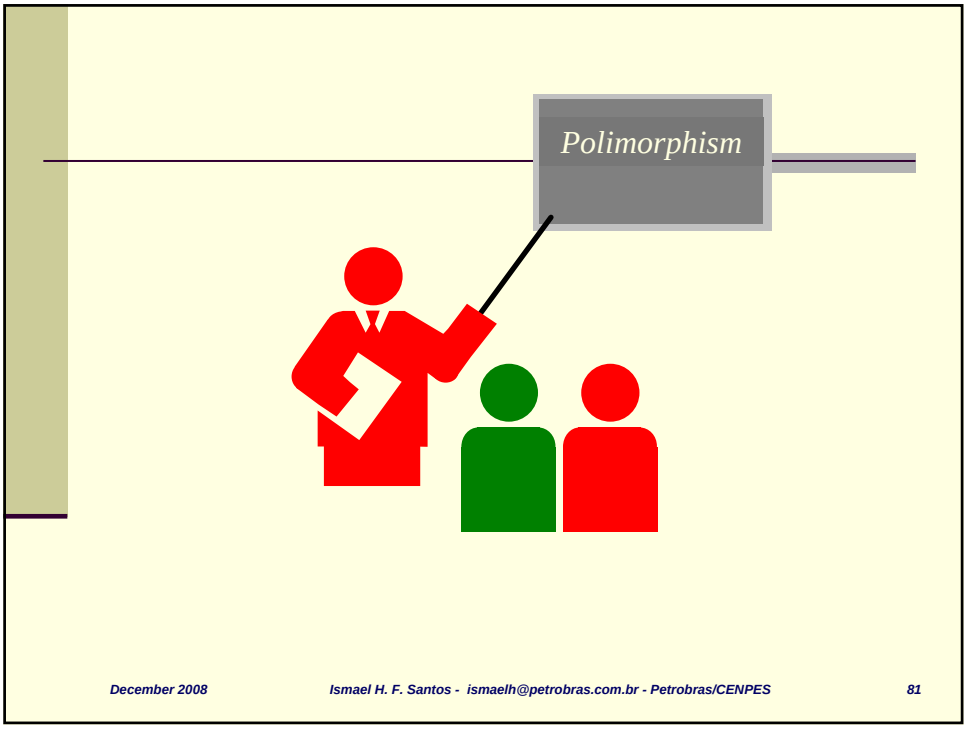
- Herança de Membros



Information Hiding

- Herança de Membros entre Pacotes





Polymorphism

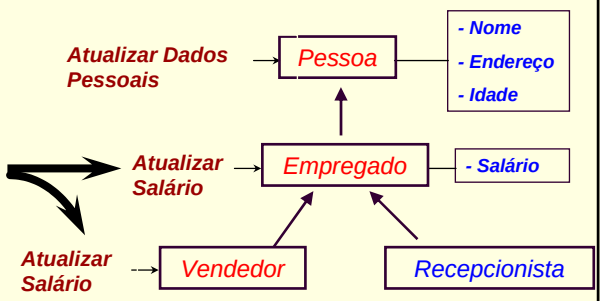
- *Polymorphism* is the capability of different objects to react in an individual manner to the same message.
- Polymorphism is widely used in natural languages. Consider the verb *to close*: It means different things when applied to different objects. Closing a door, closing a bank account, or closing a program's window are all different actions; their exact meaning depends on the object on which the action is performed.
- Similarly, polymorphism in OO programming means that the interpretation of a message depends on its object.
- C++ has three mechanisms of static (compile-time) polymorphism: operator overloading, templates, and function overloading.

Polymorphism

- **Polimorfismo** (poli=muitos, morfo=forma) é uma característica essencial de linguagens orientadas a objeto
- Como funciona?
  - Um objeto que faz papel de interface serve de **intermediário** fixo entre o programa-cliente e os objetos que irão executar as mensagens recebidas
  - O programa-cliente não precisa saber da existência dos outros objetos
  - Objetos podem ser substituídos sem que os programas que usam a interface sejam afetados

Polymorphism

- Polimorfismo é a capacidade dos objetos reagirem de diferentes maneiras para uma mesma mensagem.



- A capacidade polimórfica decorre diretamente do mecanismo de herança. Ao estendermos ou especializarmos uma classe, não perdemos compatibilidade com a superclasse.

Polymorphism - Example

```
class Point {
 private: int x_, y_;
 public:
   Point(int x, int y) { this.x_ = x; this.y_ = y; }
   void print() { std::cout<<"Point ("<<x<<","<<y_<<")";
 }
 class Pixel : public Point {
   private: int color_;
   public:
     Pixel(int x, int y, int color) : Point(x,y), color(color) {}
}
```

Com essa modificação, tanto a classe Point quanto a classe Pixel agora possuem um método que imprime o ponto representado.

```
Point pt = new Point();       // ponto em (0,0)
Pixel px = new Pixel(0,0,0);  // pixel em (0,0), color=0
pt.print();       // Imprime: "Ponto (0,0)"
px.print();       // Imprime: "Ponto (0,0)"
```


Polymorphism

- Porém, a implementação do método print não é boa para um Pixel pois a cor não é impressa. Para resolver o problema fazemos a classe Pixel redefinir o método print de forma adequada.

```
class Pixel : public Point {
        ...
 public:
 void print() {
   std.cout<<"Pixel("<<x<< ","<<y<<","<<color<<")";
 }
}
```

Com essa nova modificação, a classe Pixel agora possui um método que imprime o pixel de forma correta.

Polymorphism

- A sub-classe de Point, Pixel, é compatível com ela (Point), ou seja, um Pixel, além de outras coisas, é um ponto. Isso implica que, sempre que precisarmos de um ponto, podemos usar um Pixel em seu lugar.

```
Point[] pontos = new Point[5];   // um array de pontos
pontos[0] = new Point();
pontos[1] = new Pixel(1,2,0);    // OK! um pixel é um ponto
pontos[2] = new String("Alo");   // ERRO!, não é um ponto
```

- Note que um pixel pode ser usado sempre que se necessita um ponto. Porém, o contrário não é verdade: não podemos usar um ponto quando precisamos de um pixel.

```
Point pt = new Pixel(0,0,1);   //OK! pixel é ponto.
Pixel px = new Point(0,0);     //ERRO! ponto não é pixel.
```


Polymorphism

```
Point pt = new Point();      // ponto em (0,0)
Pixel px = new Pixel(0,0,0); // pixel em (0,0)
pt.print() ;         // Imprime: "Ponto (0,0)"
px.print();          // Imprime: "Pixel (0,0,0)"
```

- Voltemos ao exemplo do Array de pontos.

```
Point[] pontos = new Point[5];
pontos[0] = new Point();
pontos[1] = new Pixel(1,2,0);

pontos[0].print();    // Imprime: "Ponto (0,0)"
pontos[1].print();    // Imprime: "Pixel (1,2,0)"
```

Polymorphism - Example

```
class Veiculo {
private:
  char  nome_[20];
protected:
  float fuelCapacity_;
public:
  Veiculo (char* nome, float fuelCap) { strcpy_s(nome_, nome);
                                        fuelCapacity_ = fuelCap; }
  virtual ~Veiculo(void) {}
  char* getName() { return nome_; }
  float capacity(void);        // virtual float capacity(void);
  // pure virtual methods      // using early binding ...
  virtual void adjust(void)    = 0;
  virtual void checkList(void) = 0;
  virtual void cleanUp(void)   = 0;
};
```


Polymorphism - Example

```
#include "Veiculo.h"
class Motocicleta : public Veiculo {
public:
  Motocicleta(char* nome, float fuelCap) : Veiculo(nome, fuelCap){}
  virtual ~Motocicleta (void) { }
  // implmented virtual methods
  virtual float capacity() { return fuelCapacity_; }
  virtual void adjust(void){ cout<<"Verifica marcha lenta"<<endl; }
  virtual void checkList(void){cout<<"Checklist used parts"<<endl;}
  virtual void cleanUp(void){ cout<<“Washing2"<<endl; }
};

#include "Veiculo.h"
class Automovel : public Veiculo {
public:
  Automovel(char* nome, float fuelCap) : Veiculo(nome, fuelCap){}
  virtual ~ Automovel(void) { }
  // implmented virtual methods
  ..................
```

Polymorphism - Example

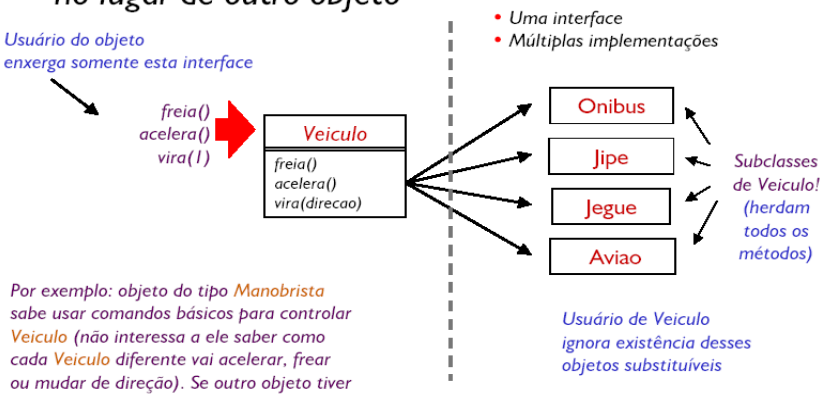
```
class Oficina {
private:
  char nome_[30];  char cgc_[10];
  int atende (Veiculo* veiculo) {
    cout<<"-> Veiculo: "<<veiculo->getName()<<" fuelCap= "<<
           veiculo->capacity()<<endl;
    veiculo->adjust(); veiculo->checkList(); veiculo->cleanUp();
    cout<<"--------------------------------"<<endl;  return 1;
  }
public:
  Oficina(char* nome, char* cgc) { strcpy_s(nome_, sizeof(nome_),
                     nome); strcpy_s(cgc_, sizeof(cgc_), cgc); }
  virtual ~Oficina(void) { }
  int atende(Veiculo** queueVeiculo, int nrVeiculos) {
    cout<<"=== Oficina:"<<nome_<<" CGC: "<<cgc_<<"====="<<endl;
    for (int i=0; i<nrVeiculos; ++i) {
      if( atende(queueVeiculo[i]) != 1 ) { return 0; }
    }
    return 1;
  }
};
```


Polymorphism - Example

```
void main(){
  // Create a veiculo queue
  Veiculo* queue[4];
  queue[0] = new Automovel("fusca", 20);
  queue[1] = new Automovel("opala", 40);
  queue[2] = new Motocicleta("BMW", 10);
  queue[3] = new Automovel("cadillac", 50);

  // Leva veiculos para revisao
  Oficina o("Jacarezinho", "6669");
  if( o.atende(queue, 4) == 1 ) {
    cout<<"revisao OK";
  }
  else {
    cout<<"revisao NOT OK";
  }
  getchar();
}
```

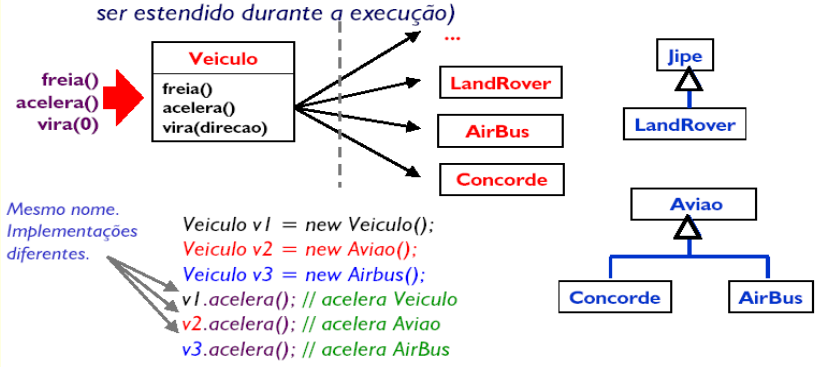
Outro exemplo - Polymorphism

- *Polimorfismo significa que um objeto pode ser usado no lugar de outro objeto*



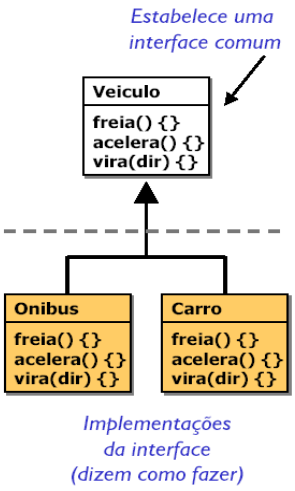
Polymorphism => Código extensível

- *Novos objetos podem ser usados em programas que não previam a sua existência*
  - *Garantia que métodos da interface existem nas classes novas*
  - *Objetos de novas classes podem ser criados e usados (programa pode ser estendido durante a execução)*



Interface x Implementação

- *Polimorfismo permite separar a interface da implementação*
- *A classe base define a interface comum*
  - *Não precisa dizer **como** isto vai ser feito*
    Não diz: eu sei como frear um Carro ou um Ônibus
  - *Diz apenas que os métodos existem, que eles retornam determinados tipos de dados e que requerem certos parâmetros*
    Diz: Veiculo pode acelerar, frear e virar para uma direção, mas a direção deve ser fornecida



Polymorphism - Late Binding

- Late Binding (Ligação tardia)
  - As linguagens OO possuem um recurso chamado late binding, que permite o adiamento da resolução de um método até o momento no qual ele deve ser efetivamente chamado. Ou seja, a resolução do método acontecerá em tempo de execução, ao invés de em tempo de compilação. No momento da chamada, o método utilizado será o definido pela classe real do objeto.
  - Voltando ao exemplo do array de pontos, agora que cada classe possui sua própria codificação para o método print, o ideal é que, ao correrrmos o array imprimindo os pontos, as versões corretas dos métodos fossem usadas. Isso realmente acontece, pois as linguagens OO usam um recurso chamado late binding.

Late Binding na prática

- Graças a esse recurso, agora temos:

```
Point[] pontos = new Point[5];
pontos[0] = new Point();
pontos[1] = new Pixel(1,2,0);
pontos[0].print();    // Imprime: “Point
  (0,0)”
pontos[1].print();    // Imprime: “Pixel
  (1,2,0)”
```

- Suporte ao polimorfismo depende do suporte à ligação tardia (late binding) de chamadas de função

Late Binding na prática

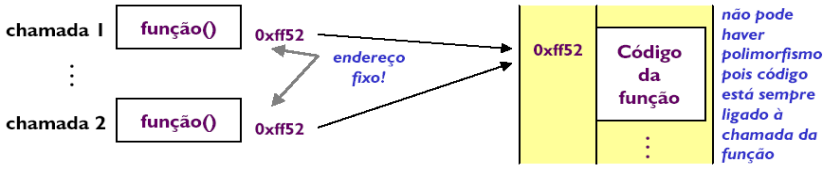
- Late Binding x Eficiência
  - O uso de late binding implica em perda no desempenho dos programas visto que a cada chamada de método um processamento adicional deve ser feito, devido ao uso da indereção causada pelo mecanismo de implementação do late-binding (tabelas métodos virtuais). Esse fato levou várias linguagens OO a permitir a construção de métodos constantes (chamados métodos finais em Java), ou seja, métodos cujas implementações não podem ser redefinidas nas sub-classes.

Late Binding => suporte Polymorphism

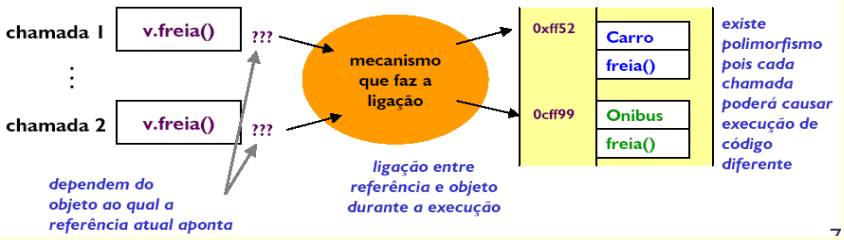
- *Suporte a polimorfismo depende do suporte à ligação tardia (late binding) de chamadas de função*
  - *A referência (interface) é conhecida em **tempo de compilação** mas o objeto a que ela aponta (implementação) não é*
  - *O objeto pode ser da mesma classe ou de uma subclasse da referência (garante que a TODA a interface está implementada no objeto)*
  - *Uma única referência, pode ser ligada, **durante a execução**, a vários objetos diferentes (a referência é polimorfa: pode assumir muitas formas)*

Late x Early Binding

- *Em tempo de compilação (early binding) - C!*

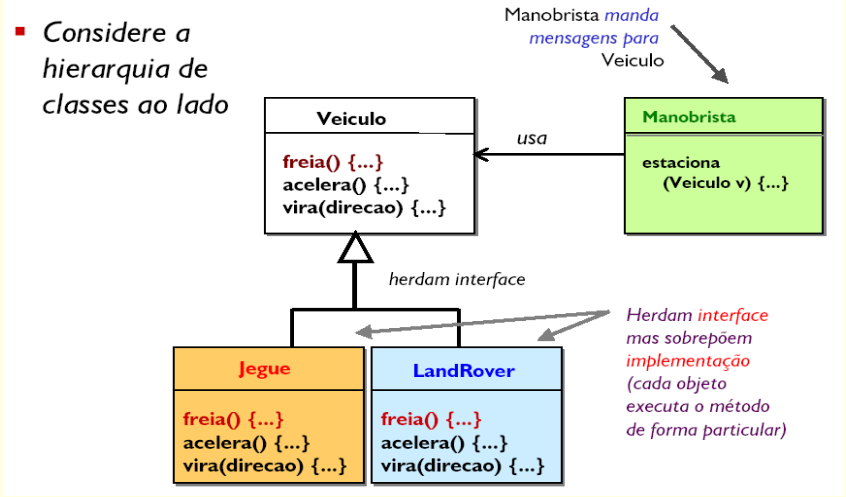


- *Em tempo de execução (late binding) - Java!*



Outro exemplo - Polimorfismo

- *Considere a hierarquia de classes ao lado*

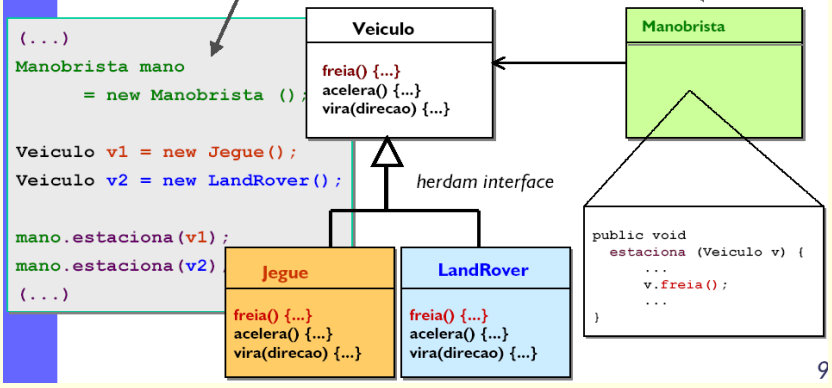


Outro exemplo - Polimorfismo

Trecho de programa que usa Manobrista: Em tempo de execução passa implementação de Jegue e LandRover no lugar da implementação original de Veiculo (aproveita apenas a interface de Veiculo)

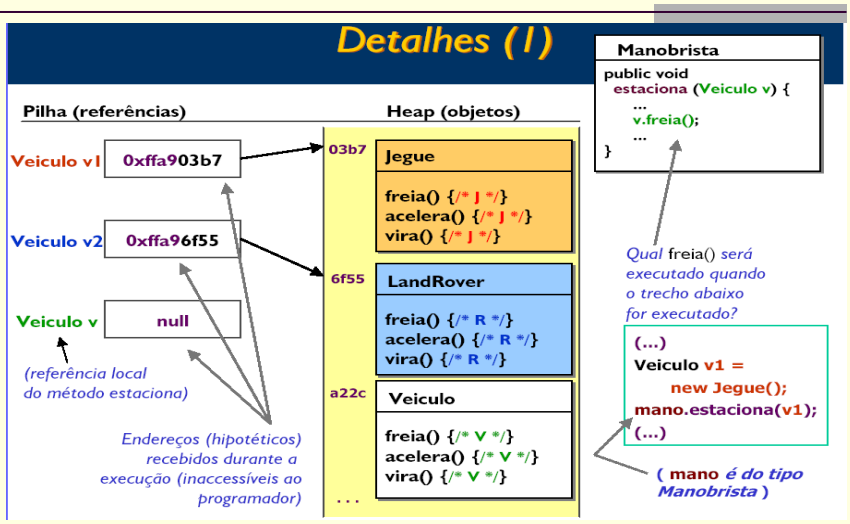
Manobrista usa a classe Veiculo (e ignora a existência de tipos específicos de Veiculo como Jegue e LandRover

```
(...)
Manobrista mano
      = new Manobrista ();

Veiculo v1 = new Jegue();
Veiculo v2 = new LandRover();

mano.estaciona(v1);
mano.estaciona(v2);
(...)
```

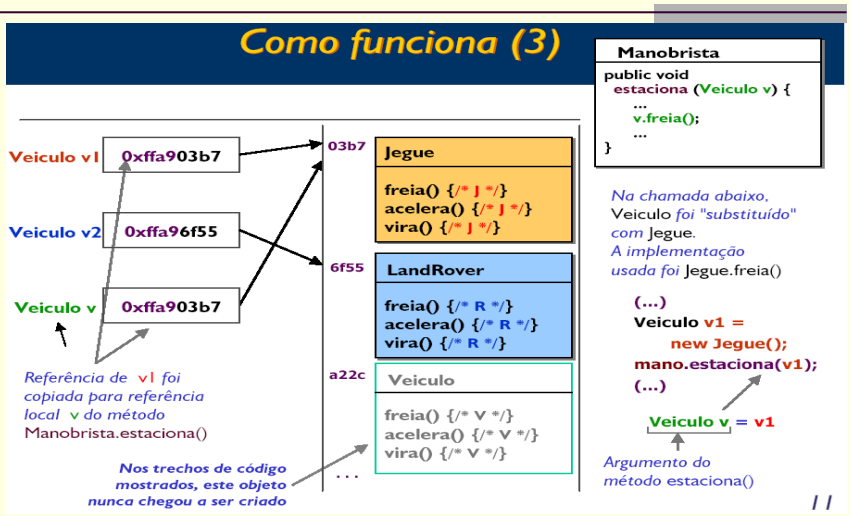


9

Outro exemplo - Polimorfismo



Outro exemplo - Polimorfismo



Alguns comentários

- *Como deve ser implementado freia() na classe Veiculo?*
  - *Faz sentido dizer como um veículo genérico deve frear?*
  - *Como garantir que cada tipo específico de veículo redefina a implementação de freia()?*
- *O método freia() é um procedimento abstrato em Veiculo*
  - *Deve ser usada apenas a implementação das subclasses*
- *E se não houver subclasses?*
  - *Como freia um Veiculo genérico?*
  - *Com que se parece um Veiculo generico?*
- *Conclusão: não há como construir objetos do tipo Veiculo*
  - *É um conceito genérico demais*
  - *Mas é ótimo como interface! Eu posso saber dirigir um Veiculo sem precisar saber dos detalhes de sua implementação*

Paradigma OO - Sumário

- Agentes são objetos;
- Ações são executadas através da troca de mensagens entre objetos;
- Todo objeto é uma instância de uma classe;
- Uma classe define uma interface e um comportamento;
- Classes podem estender outras classes através de herança.

Paradigma OO - Sumário

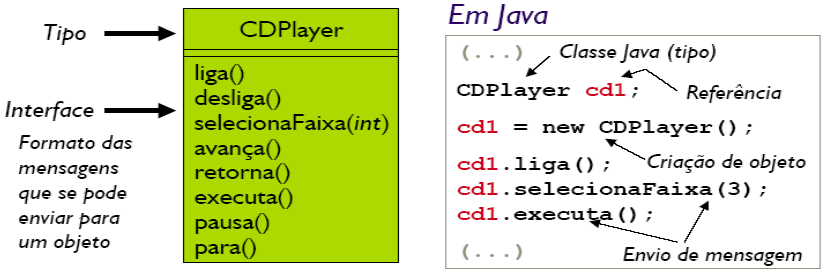
- Objetos são **conceitos** que têm
  - identidade,
  - estado e
  - comportamento
- Características de Smalltalk, resumidas por Allan Kay:
  - Tudo (em um programa OO) são objetos
  - Um programa é um monte de objetos enviando mensagens uns aos outros
  - O espaço (na memória) ocupado por um objeto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

Paradigma OO - Sumário

- 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

Sumário - Objetos

- 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

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

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

Sumário - Objetos

- Implementação não interessa à quem **usa** objetos
- Papel do 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 do 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

Sumário - Objetos

- *Os componentes de uma classe, em Java, podem pertencer a dois domínios, que determinam como são 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 classes que usam os objetos*

Paradigma OO

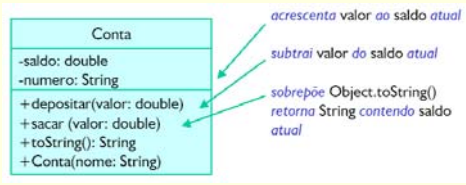
- Por que OO ?

- **Porque promove o desenvolvimento de software com qualidade**
  - **Correção**
  - **Robusto**
  - **Extensibilidade**
  - **Reusabilidade**
  - **Compatibilidade**

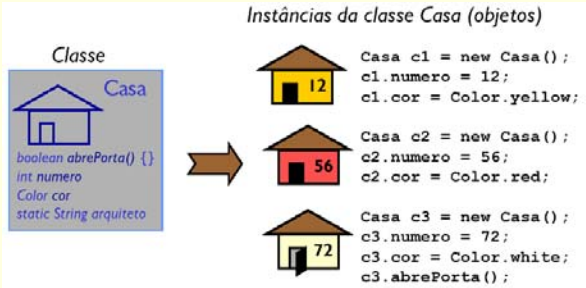
Notação UML

Notação UML

- Classes são uma especificação para objetos, representa um tipo de dados complexo.

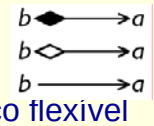


- Objetos são instancias da classe

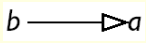


Notação UML (cont.)

- Separação interface-implementação: permite um 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***

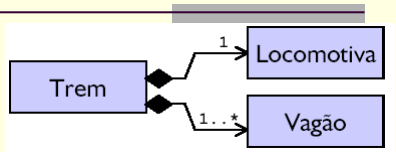


- Reuso da interface da classe: pouco flexível
  - ***Herança: b "é" a (substituição pura) ou b "é um tipo de" a (substituição útil, extensão)***

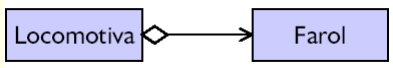


Notação UML (cont.)

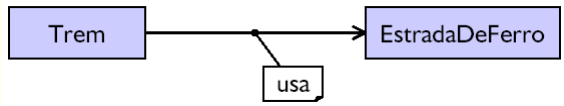
- *Composição*: ***um trem é formado por locomotiva e vagões***



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

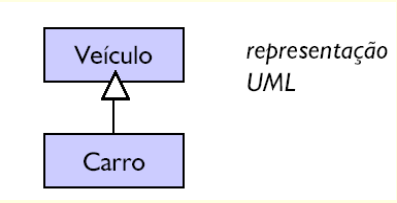


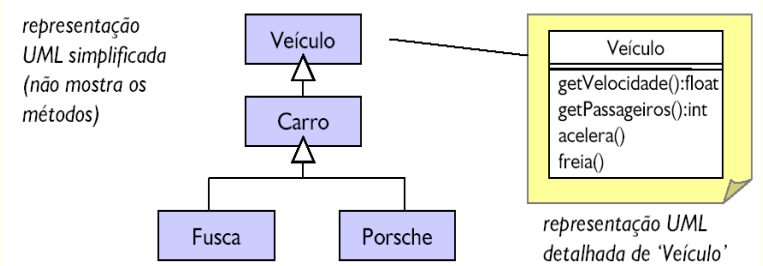
- *Associação*: ***um trem usa uma estrada de ferro (não faz parte do trem, mas ele depende dela)***



Notação UML (cont.)

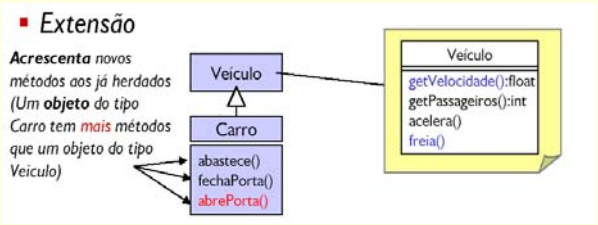
- Herança
  - Um carro é um veiculo
  - Fuscas e Porsches são carros (e também são veículos)



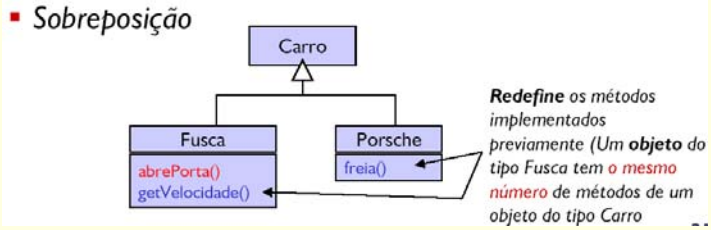


Notação UML (cont.)

- Extensão

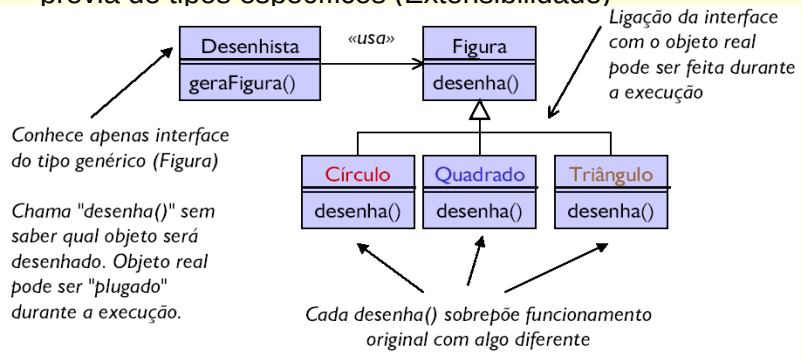


- Sobreposição



Notação UML (cont.)

- Polimorfismo
  - Uso de um objeto no lugar de outro. Dessa forma pode-se escrever código que não dependa da existência previa de tipos específicos (Extensibilidade)



Sumário 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"*
  - *Consiste na delegação de operações a outros 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 objeto seja usado no lugar de outro*

End of Module II