Paradigmas de Linguagens de Programação

Programação Orientada a Objetos

Introdução

- Linguagens que suportam **programação orientada a objetos (POO)** são amplamente utilizadas.
- C++, Objective-C, Java e C# são exemplos de linguagens **imperativas com suporte à orientação a objetos**.
- Outras linguagens, como **CLOS** (LISP) e **OCaml**, unem **programação funcional e orientada a objetos**.
- Ruby é uma linguagem **híbrida**, pois permite programação orientada a objetos e também procedural.
- Smalltalk é o exemplo clássico de **linguagem puramente orientada a objetos**.

Introdução

- POO aplica o **princípio da abstração** a coleções de tipos de dados semelhantes.
- O conceito central é **herança:**
  - membros comuns são colocados em um tipo “pai”, permitindo que tipos derivados compartilhem atributos e comportamentos.

Introdução

- A **herança** permite **reuso e especialização** de tipos.
- A **vinculação dinâmica de métodos** torna o comportamento mais flexível.
- Métodos são chamados dinamicamente em tempo de execução, permitindo **polimorfismo** (um mesmo nome de método executa ações diferentes conforme o tipo do objeto).
- O **encapsulamento** protege os dados internos, expondo apenas as operações permitidas.

Programação orientada a objetos

- O conceito de **POO** tem raízes no **SIMULA 67**, mas foi consolidado com o **Smalltalk-80** (1980).
- Smalltalk é considerado o **modelo base** de linguagem puramente orientada a objetos.
- Uma linguagem orientada a objetos deve oferecer suporte a três mecanismos principais:
  - **Tipos abstratos de dados**
  - **Herança**
  - **Vinculação dinâmica de métodos (polimorfismo)**

Programação orientada a objetos

- Um **Tipo Abstrato de Dado (TAD)** é uma **forma de organizar dados e operações** de maneira encapsulada.
- Ele define **o que pode ser feito** (as operações), mas **oculta como é feito** (a implementação).
- Em um TAD:
  - O tipo de dado é declarado;
  - As operações válidas sobre ele são especificadas;
  - A implementação interna é escondida do usuário.

Programação orientada a objetos

- Abstração é uma **representação simplificada** de uma entidade, destacando apenas seus **atributos mais relevantes**.
- Em programação, abstração permite **enfatizar o essencial e ignorar o supérfluo**.
- Existem dois tipos principais:
  - **Abstração de processo:** focada em *como* algo é feito (ex.: subprogramas, funções).
  - **Abstração de dados:** focada em *o que* algo é (ex.: tipos de dados e operações sobre eles).
- A POO surge como uma evolução da abstração de dados, onde **dados e operações** são tratados como **unidade única: o objeto**.

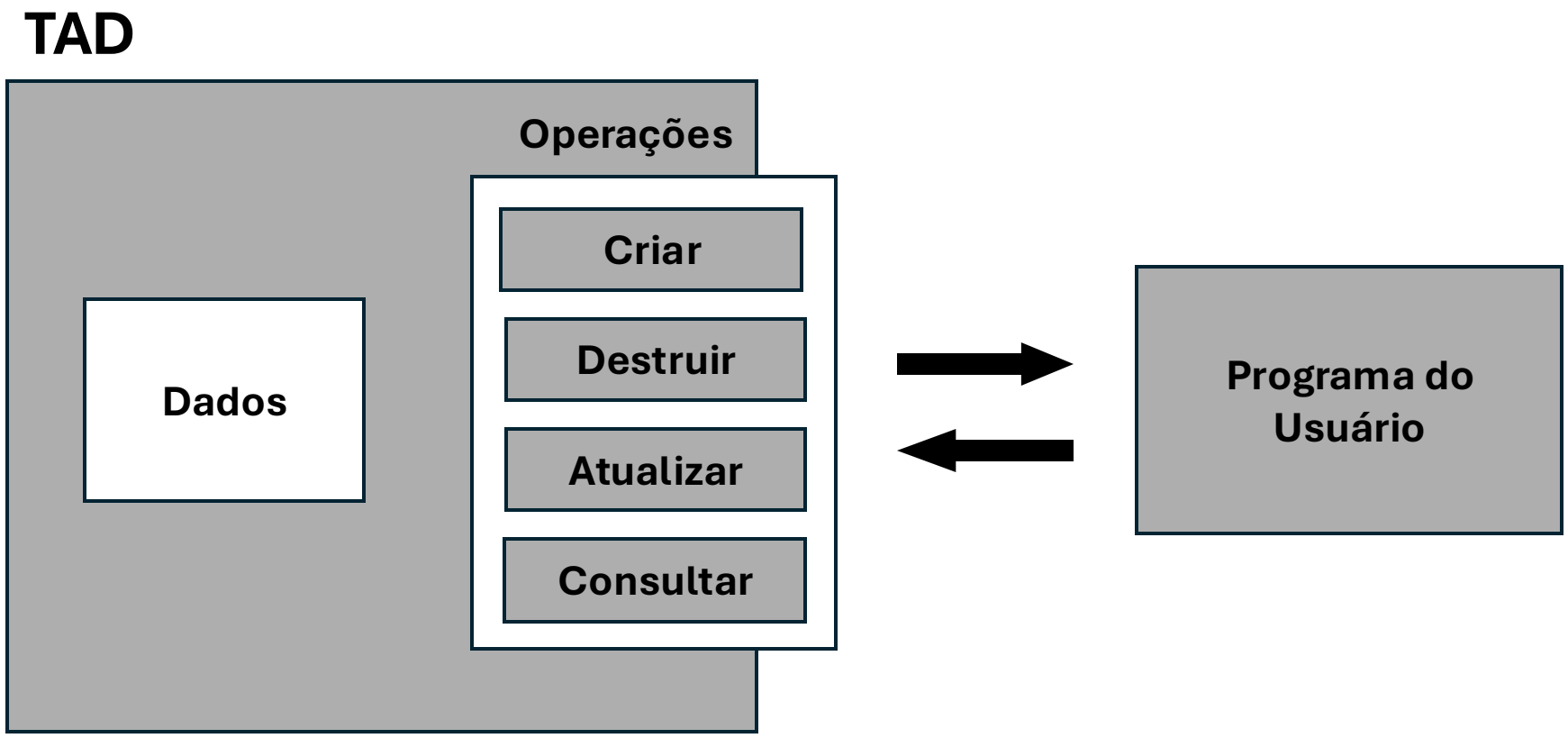
Programação orientada a objetos

- O **encapsulamento** une dados e operações em uma **única unidade sintática**.
- A **ocultação de informação** restringe o acesso direto aos dados internos:
  - O usuário interage apenas por meio dos **métodos públicos**.
  - A implementação interna pode mudar sem afetar o código cliente.
- Benefícios:
  - Reduz o impacto de mudanças.
  - Facilita o reuso e a manutenção.
  - Evita conflitos de nomes e acessos indevidos.

Programação orientada a objetos

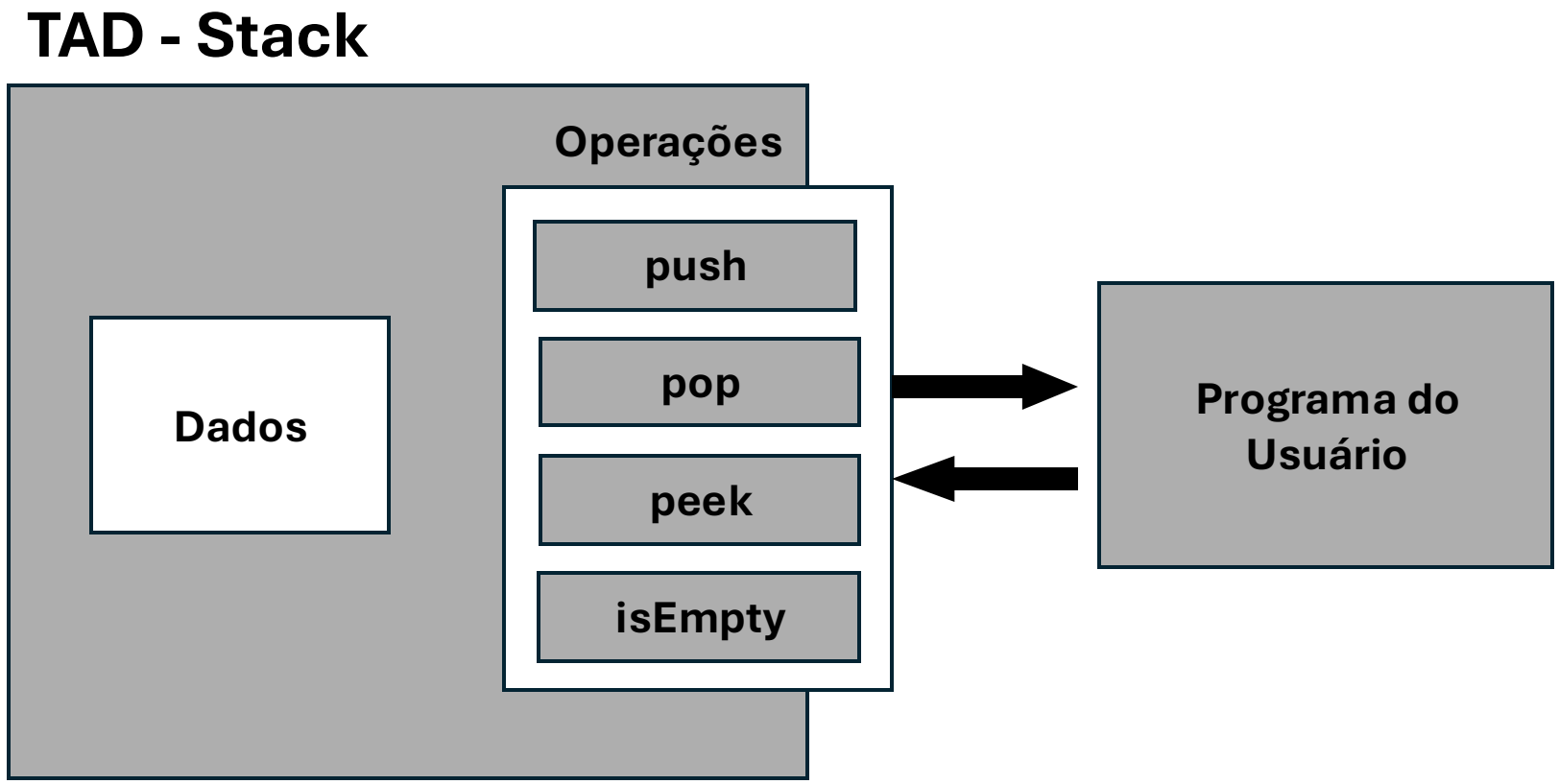
- Em um **TAD**, o acesso aos componentes é **controlado** por níveis de visibilidade.
  - **public:** indica as **operações** que podem ser usadas pelo programa do usuário (interface pública).
  - **private:** indica os **dados internos** e detalhes da implementação, **ocultos do usuário**.
- Essa separação reforça o **encapsulamento** e a **ocultação de informação**.
- O usuário interage apenas com as operações públicas, **sem saber como os dados são implementados**.

Programação orientada a objetos



Programação orientada a objetos

- Exemplo **pilha (stack)**
- Operações: push, pop, peek, isEmpty.
- O usuário não precisa saber se a pilha é implementada com vetor ou lista encadeada.



Programação orientada a objetos

- Desde o final dos anos 1980, buscava-se **aumentar a produtividade** através do **reuso de software**.
- **Tipos abstratos de dados (TADs)**, com encapsulamento e controle de acesso, pareciam ideais para reuso.
- Porém, o reuso direto de TADs é limitado:
  - O tipo existente raramente se ajusta exatamente ao novo uso.
  - Pequenas modificações exigem entender e alterar código legado.
  - Alterações em um tipo reutilizado podem afetar todos os programas que o utilizam.
- Outro problema: **TADs são independentes e estão no mesmo nível hierárquico**, o que dificulta combinar suas definições de forma organizada.

Programação orientada a objetos

Herança

- A **herança** resolve o problema de reuso e organização de tipos.
- Permite que um novo tipo **herde dados e operações** de outro tipo já existente.
- O novo tipo pode **modificar** ou **acrescentar** novos recursos ao tipo herdado.
- Dessa forma:
  - O reuso é facilitado.
  - A estrutura do programa reflete relações naturais de descendência entre entidades.
- A herança fornece um **framework hierárquico** que modela relacionamentos de “pai e filho” entre classes.

Programação orientada a objetos

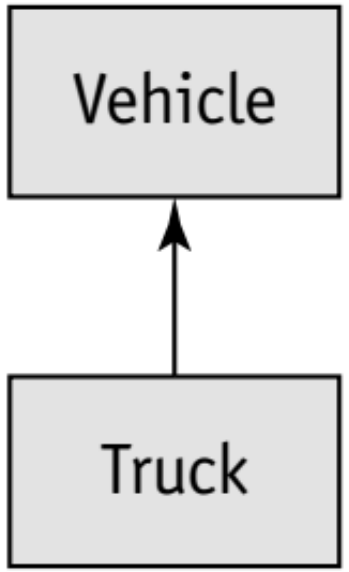
Herança

- Em linguagens orientadas a objetos, os TADs passam a ser chamados de **classes**.
- As instâncias das classes são os **objetos**.
- Uma classe derivada de outra é chamada de:
  - **Subclasse**, **classe derivada** ou **classe filha**.
- A classe da qual ela deriva é chamada de:
  - **Superclasse**, **classe base** ou **classe pai**.
- As operações associadas aos objetos são chamadas de **métodos**.
- O conjunto de métodos de uma classe constitui seu **protocolo de mensagens/interface de mensagens**.

Programação orientada a objetos

Herança

- Suponha uma classe *Vehicle* com variáveis como **marca**, **cor** e **modelo**.
- Uma subclasse *Truck* poderia herdar essas variáveis e adicionar novas, como:
  - **capacidade de reboque**
  - **número de rodas**



Programação orientada a objetos

Herança

- A subclasse pode adicionar variáveis e/ou métodos àqueles herdados da classe pai.
- A subclasse pode modificar o comportamento de um ou mais métodos herdados.
- A classe pai pode definir que algumas de suas variáveis ou métodos têm acesso privado, o que significa que não serão visíveis na subclasse.

Programação orientada a objetos

Herança

- Um **método sobrescrito** redefine o comportamento de um método herdado.
- A nova versão substitui a anterior apenas para objetos da subclasse.
- Exemplo
  - A subclasse *Dog* **sobrescreve** o método *makeSound()* herdado de *Animal*.

```
class Animal {
   void makeSound()

class Dog extends Animal{
   void makeSound()
}
```

Programação orientada a objetos

Herança

- As classes possuem dois tipos principais de variáveis e métodos:
- **Variáveis de instância:** Pertencem a cada objeto, cada instância tem sua cópia.
- **Variáveis de classe:** Compartilhados entre todas as instâncias da classe.
- **Métodos de instância:** operam sobre variáveis específicas de cada objeto.
- **Métodos de classe:** operam sobre variáveis comuns a toda a classe.

Programação orientada a objetos

Herança

- Se uma subclasse deriva de apenas uma classe pai: **herança simples**.
- Se deriva de múltiplas classes pai: **herança múltipla**.
- A **herança múltipla** permite combinar funcionalidades de várias classes, mas pode gerar **ambiguidade e dependências complexas**.

Programação orientada a objetos

Herança e encapsulamento

- A herança introduz um **terceiro nível de acesso**, além de *public* e *private*:
  - **Privado (private):** visível apenas na **classe base**.
  - **Público (public):** visível para **todos**.
  - **Protegido (protected):** visível para **subclasses**, mas **não para clientes externos**.
- O encapsulamento protege os dados internos da classe:
  - Membros **privados** não são visíveis nas subclasses.
  - Membros **públicos** são herdados normalmente.
  - Membros **protegidos** são herdados, mas só acessíveis dentro da hierarquia.

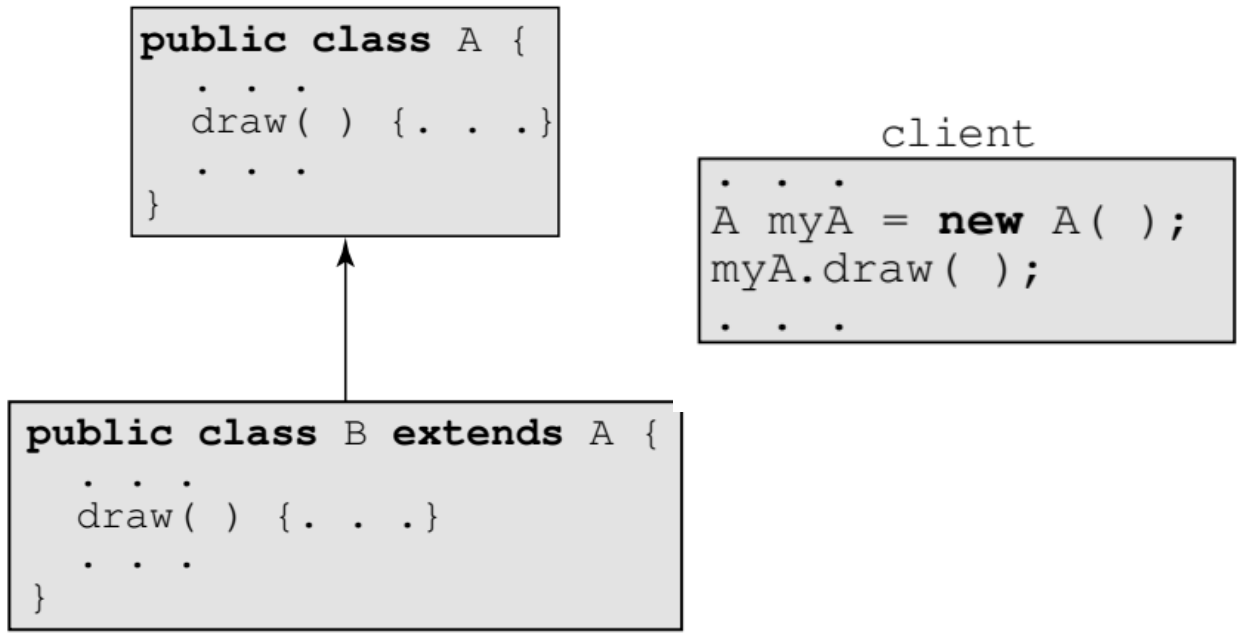
Programação orientada a objetos

Vinculação dinâmica

- A **vinculação dinâmica** é a terceira característica essencial das linguagens que suportam programação orientada a objetos, junto aos **tipos de dados abstratos** e à **herança**.
- Fornece um tipo de **polimorfismo**, chamado também de **despacho dinâmico**.
- A vinculação dinâmica de métodos é o processo pelo qual o **método a ser executado é determinado em tempo de execução**, e não em tempo de compilação.
- Isso permite que objetos de diferentes classes relacionadas respondam a mensagens de maneira apropriada aos seus próprios tipos.

Programação orientada a objetos

Vinculação dinâmica




```
public class A {
   void draw() { System.out.println("A"); }
}

public class B extends A {
   void draw() { System.out.println("B"); }
}

// Cliente
A myA = new A();
A myB = new B();
myA.draw(); // Desenha A
myB.draw(); // Desenha B
```

Programação orientada a objetos

Vinculação dinâmica e classes abstratas

- Em alguns casos, uma hierarquia de herança pode incluir classes cujas instâncias **não fazem sentido por si mesmas**.
- Por exemplo, uma classe *Building* usada como base para subclasses como *French_Gothic_Cathedrals*.
- Nesse caso, Building pode declarar um **método abstrato** draw() sem corpo.
- Uma classe que contenha pelo menos um método abstrato é chamada de **classe abstrata**, e normalmente não pode ser instanciada.
- Qualquer classe derivada deve **fornecer implementações concretas** para os métodos abstratos herdados.

Questões de projeto para linguagens OO

- Ao projetar linguagens com suporte à **herança** e **vinculação dinâmica**, surgem diversas decisões de projeto.
- As principais questões tratadas incluem:
  - Exclusividade dos objetos
  - Subclasses como subtipos
  - Herança simples e múltipla
  - Alocação e liberação de objetos
  - Vinculação estática e dinâmica
  - Classes aninhadas
  - Inicialização de objetos

Questões de projeto para linguagens OO

Exclusividade dos objetos

- Um projetista totalmente comprometido com o modelo de objetos faz com que **todos os tipos sejam objetos**, até inteiros escalares.
- **Vantagens:** elegância, uniformidade e simplicidade de uso.
- **Desvantagem:** operações simples se tornam mais lentas, pois exigem passagem de mensagens em vez de instruções diretas.
- Alternativas:
  - Manter **tipos primitivos separados** e adicionar o modelo de objetos por cima.
  - Tratar **todos os tipos compostos como objetos**, mas manter **tipos primitivos como valores simples**.

Questões de projeto para linguagens OO

As subclasses são subtipos?

- Uma linguagem suporta o **princípio de substituição** se uma variável de uma classe pai puder ser substituída por uma variável de uma classe derivada **sem causar erros de tipo**.
- Exemplo: se B é derivada de A, então B é um **subtipo** de A.
- Em geral, **subclasses são subtipos**, a menos que o projetista as defina de modo a alterar seu comportamento.
- Linguagens como Ada permitem definir subtipos explicitamente:

  subtype Small_Int is Integer range -100 .. 100;

- Subclasses também podem restringir membros ou comportamentos da classe pai.

Questões de projeto para linguagens OO

As subclasses são subtipos?

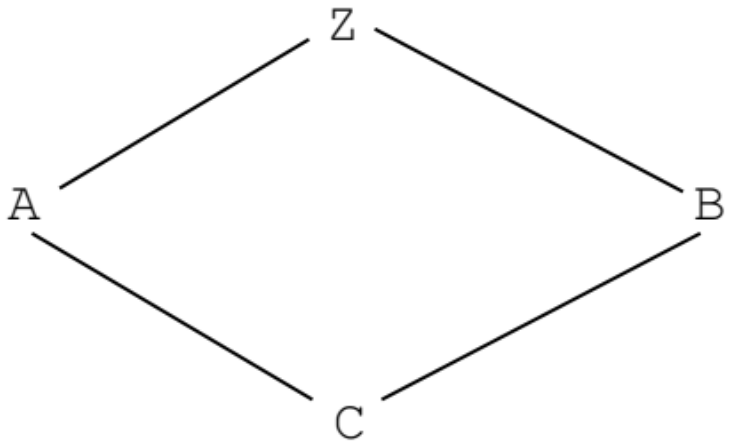
- Princípio da substituição de Liskov:
- “Se B é derivada de A, então objetos de B podem ser usados onde A é esperado, sem alterar o comportamento do programa.”
- Isso define que uma **subclasse também é um subtipo**, se o comportamento for compatível.

```
class A {
   void speak() { System.out.println("A"); }
}

class B extends A {
   void speak() { System.out.println("B"); }
}

A obj = new B();
obj.speak();  // imprime "B” substituição válida
```

Questões de projeto para linguagens OO

Herança simples e múltipla

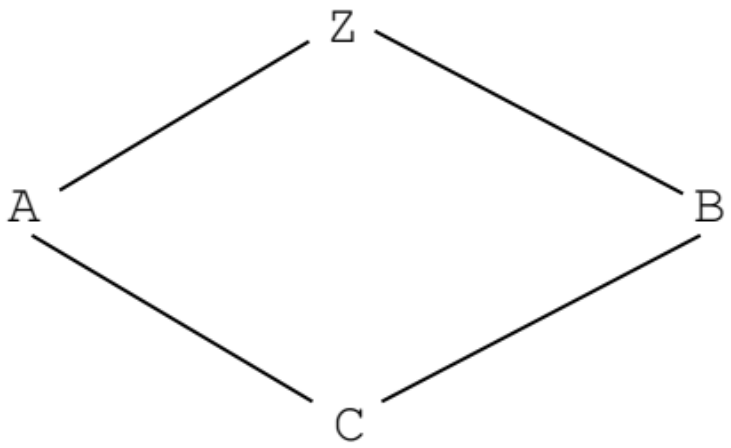
- A questão: a linguagem permite que uma nova classe **herde de duas ou mais classes**?
- **Herança múltipla** pode ser útil, mas aumenta a **complexidade e ambiguidade**.
- Exemplo clássico: **herança diamante**
- Se C herda de A e B, e ambos têm o método *display()*, qual versão deve C usar?



Questões de projeto para linguagens OO

Herança simples e múltipla

- Em C++, o problema é resolvido com **herança compartilhada** (shared inheritance).
- **Java** e **C# proíbem** herança múltipla entre classes, mas permitem múltiplas **interfaces**



Questões de projeto para linguagens OO

Interfaces como alternativa

- O uso de **herança múltipla** pode gerar programas difíceis de manter.
- **Interfaces** surgem como alternativa:
  - Oferecem as vantagens da herança múltipla (reuso e flexibilidade),
  - Evitam problemas de ambiguidade.
- Interfaces são semelhantes a **classes abstratas**, mas **não contêm implementação,** apenas declarações de métodos.

Questões de projeto para linguagens OO

Interfaces como alternativa

- Exemplo:

```
abstract class Animal {
   abstract void emitirSom();  // sem corpo
   void dormir() { System.out.println("Zzz..."); }
}

interface Voador {
   void voar();
}

class Pato extends Animal implements Voador {
   void emitirSom() { System.out.println("Quack!"); }
   public void voar() { System.out.println("Voando!"); }
}
```

Questões de projeto para linguagens OO

Alocação e liberação de objetos

- Existem duas questões de projeto principais:
  - Onde os objetos são alocados.
  - Como são liberados.
- Objetos podem ser alocados:
  - Na pilha de tempo de tempo de execução.
  - No monte (heap) criados explicitamente com um operador como o *new*.
- Quanto à liberação de memória:
  - Pode ser **implícita** (coleta automática de lixo).
  - Ou **explícita** (feita pelo programador com delete ou equivalente).

Questões de projeto para linguagens OO

Vinculação estática e dinâmica

- A **vinculação dinâmica** associa chamadas a métodos em tempo de execução.
- Alternativa: permitir que o usuário especifique se a vinculação deve ser **estática** ou **dinâmica**.
- Vantagem da estática: desempenho.
- Vantagem da dinâmica: flexibilidade e extensibilidade do sistema.
- Questão de projeto: **vale o custo de performance da vinculação dinâmica?**

Questões de projeto para linguagens OO

Classes aninhadas

- Uma **classe aninhada** é definida **dentro de outra classe**.
- Objetivo: **ocultação de informação,** se uma classe é usada apenas por outra, não precisa ser visível externamente.
- Questões de projeto:
  - Quais membros da classe externa são visíveis na aninhada?
  - Quais membros da aninhada são visíveis na classe externa?

Questões de projeto para linguagens OO

Inicialização de objetos

- A questão da **inicialização** diz respeito a **se** e **como** os objetos são inicializados com valores ao serem criados.
- Os objetos devem ser inicializados **manualmente** ou por algum **mecanismo implícito**?
- Quando um objeto de uma **subclasse** é criado, a inicialização dos membros herdados da classe pai pode ser:
  - Implícita: feita automaticamente pela linguagem.
  - Explícita: o programador precisa chamar o construtor da classe pai.

Suporte para programação orientada a objetos em linguagens específicas

- As linguagens de programação **implementam o modelo de objetos de formas diferentes**, refletindo escolhas de projeto distintas.
- Sebesta analisa como as linguagens oferecem suporte aos três recursos fundamentais:
  - Tipos de dados abstratos
  - Herança
  - Vinculação dinâmica de chamadas a métodos
- As diferenças entre essas linguagens mostram a **evolução da orientação a objetos** e suas **variações de implementação**.

Suporte para programação orientada a objetos em linguagens específicas

- Smalltalk
- **C++**
- Objective-C
- **Java**
- C#
- Ruby

C++

- O **C++** evoluiu a partir de **C** e **SIMULA 67**, com o objetivo de **suportar a programação orientada a objetos**, mantendo **compatibilidade quase completa com C**.
- C++ foi a **primeira linguagem amplamente usada** com suporte completo à orientação a objetos.
  - Ainda é uma das linguagens **mais populares**.
- Para manter compatibilidade com C, o C++ **mantém seu sistema de tipos** e adiciona **classes**.
  - **linguagem híbrida** (procedural + orientada a objetos).
- Suporta **funções e métodos independentes de classes**.

C++

- Os objetos/variáveis podem ser:
  - **Estáticos:** liberados no fim da execução
  - **Dinâmicos da pilha:** liberados automaticamente ao sair do escopo
  - **Dinâmicos do monte (heap):** precisam de liberação explícita com *delete*
- A liberação manual é necessária somente para objetos do heap, pois o C++ **não possui** recuperação de armazenamento implícita (**coleta de lixo automática**).
- Muitas classes incluem um **método destrutor**, chamado quando o objeto deixa de existir.
  - Usado para **liberar memória** ou **registrar o encerramento** do objeto.

C++

Herança

- Uma classe C++ pode ser **derivada de outra classe existente**, chamada **classe base**.
- Ao contrário de Smalltalk e de muitas linguagens orientadas a objetos, **uma classe C++ pode existir de forma independente**, sem classe base.
- Sintaxe geral:

```
class nome_classe_derivada : nome_classe_base {
   ...
};
```

C++

Herança

- Dados de uma classe são chamados de **membros de dados (data members)**, e suas funções são chamadas de **funções membro** (member functions).
- Uma classe derivada pode **herdar** membros da classe base e também **adicionar novos membros** ou **modificar** funções herdadas.

C++

Herança

- Todos os **objetos C++** devem ser **inicializados antes de serem usados**.
- Por isso, **toda classe C++** inclui pelo menos um **método construtor**.
- Construtores são **chamados implicitamente** quando o objeto é criado.
- Se a classe for derivada, o **construtor da classe base** é chamado automaticamente.
- Sintaxe comum para construtores encadeados:

```
Subclasse(parâmetros) : ClasseBase(parâmetros) {
  ...
}
```

C++

Herança

- A herança pode ser **pública**, **protegida** ou **privada**:

  class classe_derivada **: public** classe_base { ... };

  class classe_derivada **: protected** classe_base { ... };

  class classe_derivada **: private** classe_base { ... };

- **Derivação pública**
  - membros públicos e protegidos da classe base **permanecem acessíveis** na derivada.
- **Derivação protegida**
  - membros públicos e protegidos da base tornam-se **protegidos** na derivada.
- **Derivação privada**
  - membros públicos e protegidos da base tornam-se **privados** na derivada.

C++

Herança

- Exemplo
- Em subclass_1 (herança pública):
  - b e y são **protegidos**, c e z são **públicos**.
- Em subclass_2 (herança privada):
  - b, y, c e z tornam-se **privados**.

```
class base_class {
  private:
    int a;
    float x;
  protected:
    int b;
    float y;
  public:
    int c;
    float z;
};

class subclass_1 : public base_class {. . .};
class subclass_2 : private base_class {. . .};
```

C++

Herança

- Nenhuma classe derivada de subclass_2 pode acessar qualquer membro de base_class.
- Os membros a e x (privados da base) **nunca** são acessíveis, nem em subclass_1, nem em subclass_2.

```
class base_class {
  private:
    int a;
    float x;
  protected:
    int b;
    float y;
  public:
    int c;
    float z;
};

class subclass_1 : public base_class {. . .};
class subclass_2 : private base_class {. . .};
```

C++

Herança

- Na **derivação privada**, os membros públicos da classe base tornam-se **privados** na derivada.
- Nenhum membro da classe base é implicitamente visível para as instâncias da classe derivada.
- Se um membro precisar ser acessível, deve ser **reexportado**:

```
class subclass_3 : private base_class {
  base_class :: c;
  . . .
}
```

- A reexportação permite acesso controlado a membros da classe base, mantendo a derivação privada (oculta a interface da base para clientes externos).

C++

Herança

- Exemplo: classe base reutilizável
- single_linked_list representa uma lista encadeada simples.
- A classe **aninhada** node é **privada**, ocultando sua implementação das subclasses.
- As funções públicas permitem inserir, remover e testar a lista.

```
class single_linked_list {
  private:
    class node {
      public:
        node *link;
        int contents;
    };
    node *head;
  public:
    single_linked_list() {head = 0};
    void  insert_at_head(int);
    void  insert_at_tail(int);
    int   remove_at_head();
    int   empty();
};
```

C++

Herança

- stack e queue herdam publicamente de single_linked_list.
- Ambas reutilizam as operações da lista, mas **herdam toda a interface pública**.
- Isso permite que um cliente de stack chame métodos de lista como insert_at_tail(),
- o que **viola o encapsulamento** da pilha e da fila.

```
class stack : public single_linked_list {
  public:
    stack() {}
    void push(int value) {
      insert_at_head(value);
    }
    int pop() {
      return remove_at_head();
    }
};
class queue : public single_linked_list {
  public:
    queue() {}
    void enqueue(int value) {
      insert_at_tail(value);
    }
    int dequeue() {
      remove_at_head();
    }
};
```

C++

Herança

- stack_2 e queue_2 **usam a lista apenas como implementação**, sem expor sua interface pública.
- A derivação **privada** oculta métodos indesejados como insert_at_tail().
- Exemplo clássico de **reuso de implementação sem subtipagem**.

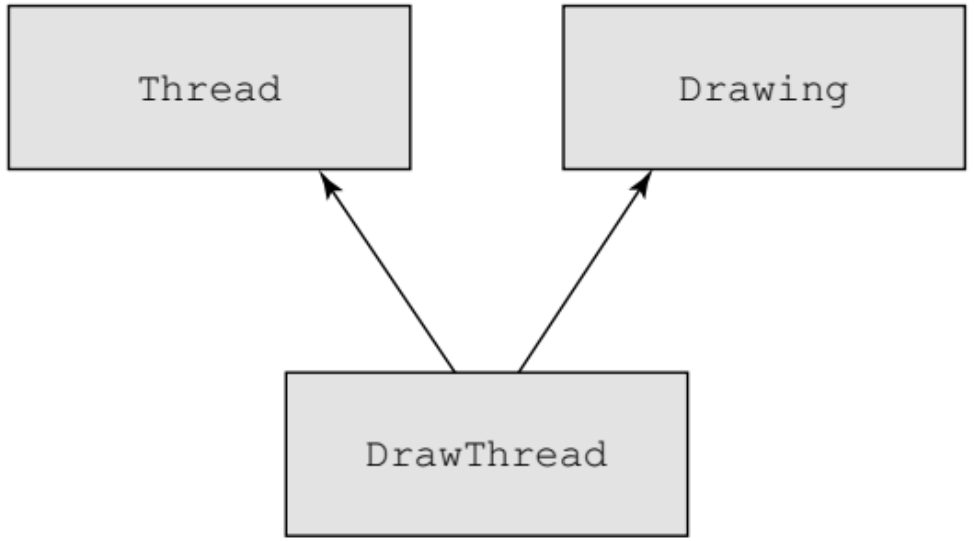
```
class stack_2 : private single_linked_list {
  public:
    stack_2() {}
    void push(int value) {
      single_linked_list :: insert_at_head(value);
    }
    int pop() {
      return single_linked_list :: remove_at_head();
    }
    single_linked_list:: empty();
};
class queue_2 : private single_linked_list {
  public:
    queue_2() {}
    void enqueue(int value) {
      single_linked_list :: insert_at_tail(value);
    }
    int dequeue() {
      single_linked_list :: remove_at_head();
    }
    single_linked_list:: empty();
};
```

C++

Herança

- O C++ permite que uma classe herde de **duas ou mais classes base (Herança múltipla)**:

```
class Thread { . . . };
class Drawing { . . . };
class DrawThread : public Thread, public Drawing { . . . };
```



C++

Herança

- O C++ permite que uma classe herde de **duas ou mais classes base (Herança múltipla)**:

```
class Thread { . . . };
class Drawing { . . . };
class DrawThread : public Thread, public Drawing { . . . };
```

- *DrawThread* herda todos os membros de *Thread* e *Drawing*.
- Se ambas tiverem membros com o mesmo nome, é preciso usar o **operador de escopo (::)** para resolver a ambiguidade.
- A herança múltipla é poderosa, mas pode gerar **problemas de ambiguidade e manutenção**.

C++

Vinculação dinâmica

- A **vinculação dinâmica** ocorre **em tempo de execução**, permitindo que uma função diferente seja chamada dependendo do tipo real do objeto.
- Para ocorrer vinculação dinâmica, o método precisa ser declarado como **virtual**.

C++

Vinculação dinâmica

- A função draw() é **virtual pura** na classe Shape
- ou seja, **não tem corpo** e **deve ser redefinida** nas classes derivadas.
- Isso torna Shape uma **classe abstrata**.

```
class Shape {
  public:
    virtual void draw() = 0;
  . . .
};
class Circle : public Shape {
  public:
    void draw() { . . . }
  . . .
};
class Rectangle : public Shape {
  public:
    void draw() { . . . }
  . . .
};
class Square : public Rectangle {
  public:
    void draw() { . . . }
  . . .
};
```

C++

Vinculação dinâmica

- Quando a chamada é feita por um **ponteiro para a classe base**, o método é escolhido **em tempo de execução:** *vinculação dinâmica*.
- Quando o objeto é conhecido (não via ponteiro), o método é resolvido **em tempo de compilação:** *vinculação estática*.

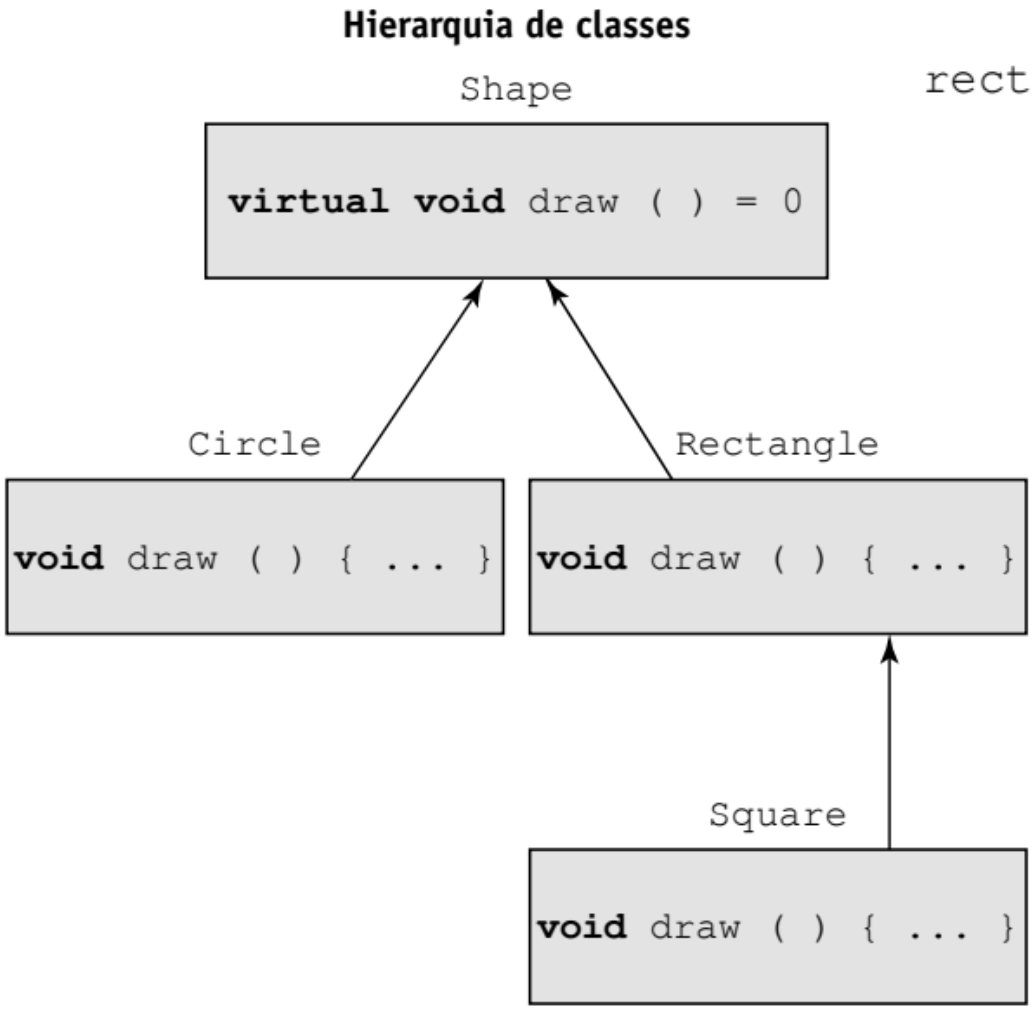
```
class Shape {
  public:
    virtual void draw() = 0;
  . . .
};
class Circle : public Shape {
  public:
    void draw() { . . . }
  . . .
};
class Rectangle : public Shape {
  public:
    void draw() { . . . }
  . . .
};
class Square : public Rectangle {
  public:
    void draw() { . . . }
  . . .
};
```

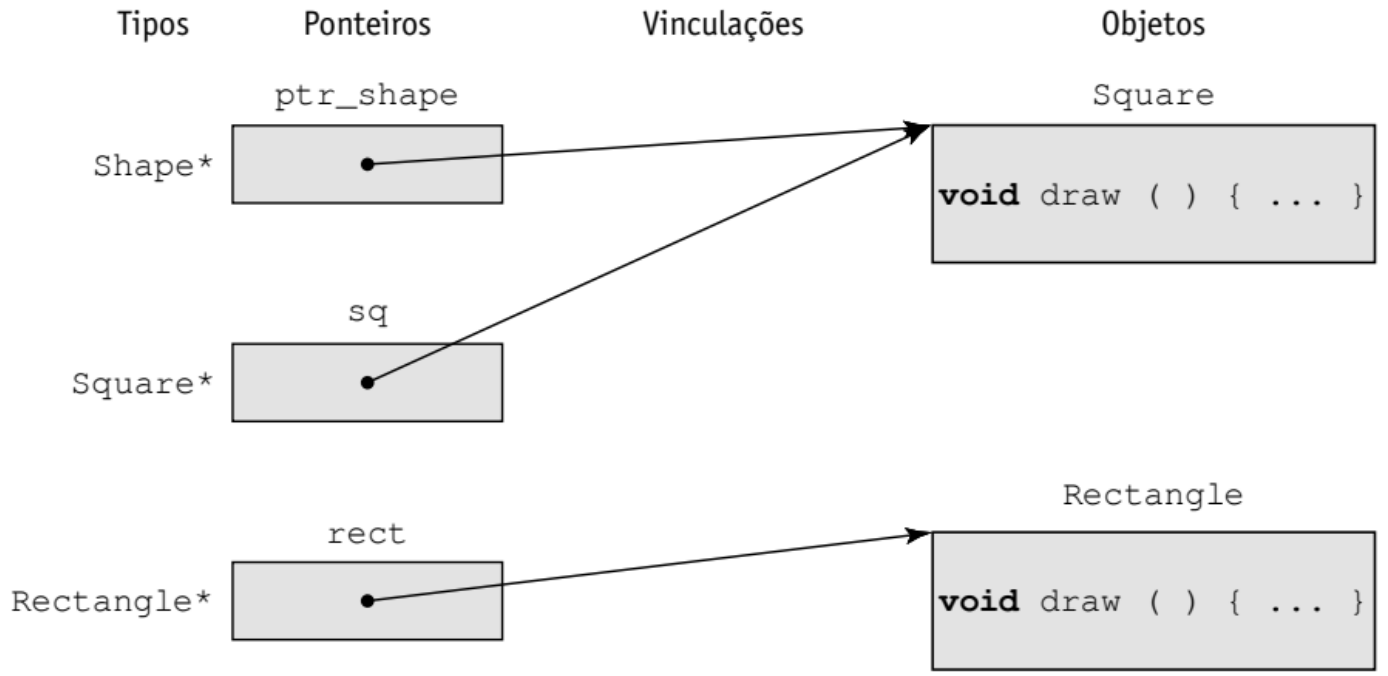
```
Square* sq = new Square;
Rectangle* rect = new Rectangle;
Shape* ptr_shape;
```

C++

Vinculação dinâmica

```
ptr_shape = sq;          // Agora ptr_shape aponta para um
                         //  objeto Square
ptr_shape->draw();       // Dinamicamente vinculado a draw
                         //  na classe Square
rect->draw();            // Estaticamente vinculado a draw
                         //  na classe Rectangle
```





C++

Vinculação dinâmica

- Uma **função virtual pura** é declarada com = 0 e **não tem corpo e nao pode ser chamada**.
- Ela **deve ser redefinida nas classes derivadas** caso seja chamada.
- Classes que contêm pelo menos uma função virtual pura são **classes abstratas**.
- Exemplo:

```
class Shape {
  public:
    virtual void draw() = 0;
   . . .
};
```

C++

Vinculação dinâmica

- C++ usa classes abstratas para **modelar hierarquias incompletas**,permitindo que subclasses definam implementações concretas.
- Assim, Shape define a interface comum, e Circle, Rectangle e Square fornecem o comportamento específico.

```
class Shape {
  public:
    virtual void draw() = 0;
  . . .
};
class Circle : public Shape {
  public:
    void draw() { . . . }
  . . .
};
class Rectangle : public Shape {
  public:
    void draw() { . . . }
  . . .
};
class Square : public Rectangle {
  public:
    void draw() { . . . }
  . . .
};
```

Java

- O modelo de classes, herança e métodos de Java é **semelhante ao de C++**.
- Java suporta **dados de objetos e não objetos**, mas apenas tipos primitivos (como int, char, boolean) **não são objetos**.
- A escolha visa **eficiência**.
- Desde o **Java 5.0**, ocorre **encaixotamento (boxing)**
  - valores primitivos são automaticamente convertidos em objetos:

```
int x = 5;
Integer obj = x;
int y = obj;
```

Java

- Em Java, **todas as classes derivam de Object**, direta ou indiretamente.
- Isso garante que métodos comuns, como toString() e equals(), sejam herdados por todas as classes.
- Em C++, por outro lado, uma classe pode existir **sem classe ancestral**.

Java

- Todos os objetos Java são **dinâmicos do monte (heap) explícitos**, criados com ***new***.
- A **coleta de lixo (garbage collection)** realiza a liberação de memória automaticamente.
- Diferente do C++, **não há operador delete** nem destrutores explícitos.

Java

- Quando o coletor de lixo está prestes a liberar um objeto, o método **finalize()** (se existir) é chamado automaticamente:

```
protected void finalize() {
    System.out.println("Objeto sendo destruído...");
}
```

- Contudo, o momento em que o coletor é executado **não é previsível**.
- Histórico recente:
  - **Java 9 (2017):** método finalize() foi **marcado como deprecated**.
  - **Java 18 (2022):** o método finalize() foi **removido oficialmente** da linguagem.
  - A recomendação atual é usar o método **AutoCloseable** com **try-with-resources** para liberar recursos de forma segura e previsível.

Java

Herança

- Um método pode ser declarado como **final**, impedindo sobrescrita em subclasses.
- Uma classe marcada como **final não pode ser herdada**.
  - Exemplo: String é uma classe final.
- Todos os métodos de uma classe **final** são implicitamente finais.
- **Vantagem:** estabilidade e segurança do comportamento.
- **Desvantagem:** impossibilidade de reutilização ou modificação.

Java

Herança

- A anotação **@Override** instrui o compilador a verificar se o método realmente sobrescreve um método ancestral.
- Se não houver método correspondente, o compilador gera erro.

Java

Herança

- O construtor da classe pai **deve ser chamado** antes do construtor da subclasse:

super(100, true);

- Se não houver chamada explícita, o compilador insere automaticamente:

super();

Java

Herança

- Java **não suporta derivação privada** (como em C++).
- Todas as subclasses em Java **são públicas** e **podem ser subtipos**.
- Java suporta **apenas herança simples**, mas inclui o tipo especial **interface** para **suporte parcial à herança múltipla**.

Java

Herança

- Java **suporta apenas herança simples**, mas inclui o tipo de classe **interface**, que fornece **suporte parcial à herança múltipla**.
- Uma **interface** é semelhante a uma classe abstrata, mas:
  - Não contém **construtores**.
  - Só pode conter **constantes** e **assinaturas de métodos** (sem corpo).
- Exemplo de definição:

```
public interface  Comparable <T> {
    public int  compareTo(T b);
}
```

- Classes que implementam a interface devem **fornecer uma implementação** para todos os métodos especificados.

Java

Herança

- Uma interface pode ser usada para **simular herança múltipla**.
- Uma classe pode **implementar várias interfaces**:

class *MyClasse* ***implements*** *InterfaceA, InterfaceB { ... }*

Java

Herança

- Além de interfaces, Java suporta **classes abstratas**, semelhantes às de C++.
- Representadas com a palavra-chave **abstract**:

```
public abstract class Figura {
    abstract void desenhar();
}
```

- Não podem ser instanciadas diretamente.
- Usadas para **modelar comportamentos comuns** em subclasses específicas.

Java

Vinculação dinâmica

- Em C++, é necessário declarar o método como **virtual** para permitir **vinculação dinâmica**.
- Em Java, **todas as chamadas a métodos são dinamicamente vinculadas**, a menos que o método seja:
  - declarado como **final**, **static** ou **private**.
- Métodos **final, static** ou **private** não podem ser sobrescritos, portanto, suas chamadas são **estaticamente vinculadas**.

Java

- O modelo de Java é **semelhante ao do C++**, mas com **maior aderência aos princípios de orientação a objetos**.
- Java **não permite classes sem pais** e usa **vinculação dinâmica por padrão**.
  - Isso torna a linguagem mais simples, mas com leve perda de desempenho em tempo de execução.
- **Interfaces** substituem a herança múltipla, fornecendo um modelo mais limpo e seguro.

Questões de projeto para linguagens OO

TABELA 12.1 Projetos

Questão de projeto/Linguagem	Smalltalk	C++	Objective-C	Java	C#	Ruby
Exclusividade de objetos	Todos os dados são objetos	Tipos primitivos, mais objetos	Tipos primitivos, mais objetos	Tipos primitivos, mais objetos	Tipos primitivos, mais objetos	Todos os dados são objetos
As subclasses são subtipos?	Podem ser e normalmente são	Podem ser e normalmente são, se a derivação for pública	Podem ser e normalmente são	Podem ser e normalmente são	Podem ser e normalmente são	Nenhuma subclasse é subtipo
Herança simples e múltipla	Apenas simples	Ambas	Apenas simples, mas alguns efeitos com protocolos	Apenas simples, mas alguns efeitos com interfaces	Apenas simples, mas alguns efeitos com interfaces	Apenas simples, mas alguns efeitos com módulos
Alocação e liberação de objetos	Todos os objetos são alocados no monte; a alocação é explícita e a liberação é implícita	Os objetos podem ser estáticos, dinâmicos da pilha ou dinâmicos do monte; a alocação e a liberação são explícitas	Todos os objetos são dinâmicos do monte; a alocação é explícita e a liberação é implícita	Todos os objetos são dinâmicos do monte; a alocação é explícita e a liberação é implícita	Todos os objetos são dinâmicos do monte; a alocação é explícita e a liberação é implícita	Todos os objetos são dinâmicos do monte; a alocação é explícita e a liberação é implícita
Vinculação dinâmica e estática	Todas as vinculações de método são dinâmicas	A vinculação de método pode ser uma ou outra	A vinculação de método pode ser uma ou outra	A vinculação de método pode ser uma ou outra	A vinculação de método pode ser uma ou outra	Todas as vinculações de método são dinâmicas
Classes aninhadas?	Não	Sim	Não	Sim	Sim	Sim
Inicialização	Construtores devem ser chamados explicitamente	Construtores são chamados implicitamente	Construtores devem ser chamados explicitamente	Construtores são chamados implicitamente	Construtores são chamados implicitamente	Construtores são chamados implicitamente

TABELA 12.1 Projetos

Questão de projeto/Linguagem	Smalltalk	C++	Objective-C	Java	C#	Ruby
Exclusividade de objetos	Todos os dados são objetos	Tipos primitivos, mais objetos	Tipos primitivos, mais objetos	Tipos primitivos, mais objetos	Tipos primitivos, mais objetos	Todos os dados são objetos
As subclasses são subtipos?	Podem ser e normalmente são	Podem ser e normalmente são, se a derivação for pública	Podem ser e normalmente são	Podem ser e normalmente são	Podem ser e normalmente são	Nenhuma subclasse é subtipo
Herança simples e múltipla	Apenas simples	Ambas	Apenas simples, mas alguns efeitos com protocolos	Apenas simples, mas alguns efeitos com interfaces	Apenas simples, mas alguns efeitos com interfaces	Apenas simples, mas alguns efeitos com módulos
Alocação e liberação de objetos	Todos os objetos são alocados no monte; a alocação é explícita e a liberação é implícita	Os objetos podem ser estáticos, dinâmicos da pilha ou dinâmicos do monte; a alocação e a liberação são explícitas	Todos os objetos são dinâmicos do monte; a alocação é explícita e a liberação é implícita	Todos os objetos são dinâmicos do monte; a alocação é explícita e a liberação é implícita	Todos os objetos são dinâmicos do monte; a alocação é explícita e a liberação é implícita	Todos os objetos são dinâmicos do monte; a alocação é explícita e a liberação é implícita
Vinculação dinâmica e estática	Todas as vinculações de método são dinâmicas	A vinculação de método pode ser uma ou outra	A vinculação de método pode ser uma ou outra	A vinculação de método pode ser uma ou outra	A vinculação de método pode ser uma ou outra	Todas as vinculações de método são dinâmicas
Classes aninhadas?	Não	Sim	Não	Sim	Sim	Sim
Inicialização	Construtores devem ser chamados explicitamente	Construtores são chamados implicitamente	Construtores devem ser chamados explicitamente	Construtores são chamados implicitamente	Construtores são chamados implicitamente	Construtores são chamados implicitamente

Paradigmas de Linguagens de Programação

Programação Orientada a Objetos