Polimorfismo e Acoplamento Dinâmico

- **Tópicos:** classificação de polimorfismo, coerção e sobrecarga, polimorfismo paramétrico, polimorfismo de inclusão e acoplamento dinâmico.
- **Objetivos:** exercitar o emprego dos diversos tipos de polimorfismo apoiados por Java.
- **Pré-requisitos:** classes e objetos, conhecimento básico sobre a linguagem Java, herança e o conceito de subtipo.

Polimorfismo

Polimorfismo (I)

- Polimorfismo significa "muitas formas" ou "tendo muitas formas"
- Funções são **polimórficas** quando seus operandos podem ter mais do que um tipo. Caso contrário, são ditas **monomórficas**
- Tipos são **polimórficos** se suas operações podem ser aplicadas para operandos de mais de um tipo. Caso contrário, são ditos **monomórficos**

Polimorfismo (II)

- **Linguagem monomórfica:** cada valor/variável só pode ser interpretado como sendo de um único tipo. Ex:

```
int i = 3;
```

- **Linguagem polimórfica:** cada valor/variável pode ser interpretado como sendo de mais de um tipo. Ex:

```
Documento d; Telegrama t; Carta c;
d = t; d = c;
d = new Telegrama();
d = new Carta();
```

Exemplo de uma Linguagem Polimórfica (I)

Função: $LENGTH :: [A] -> NUM, forall types A$

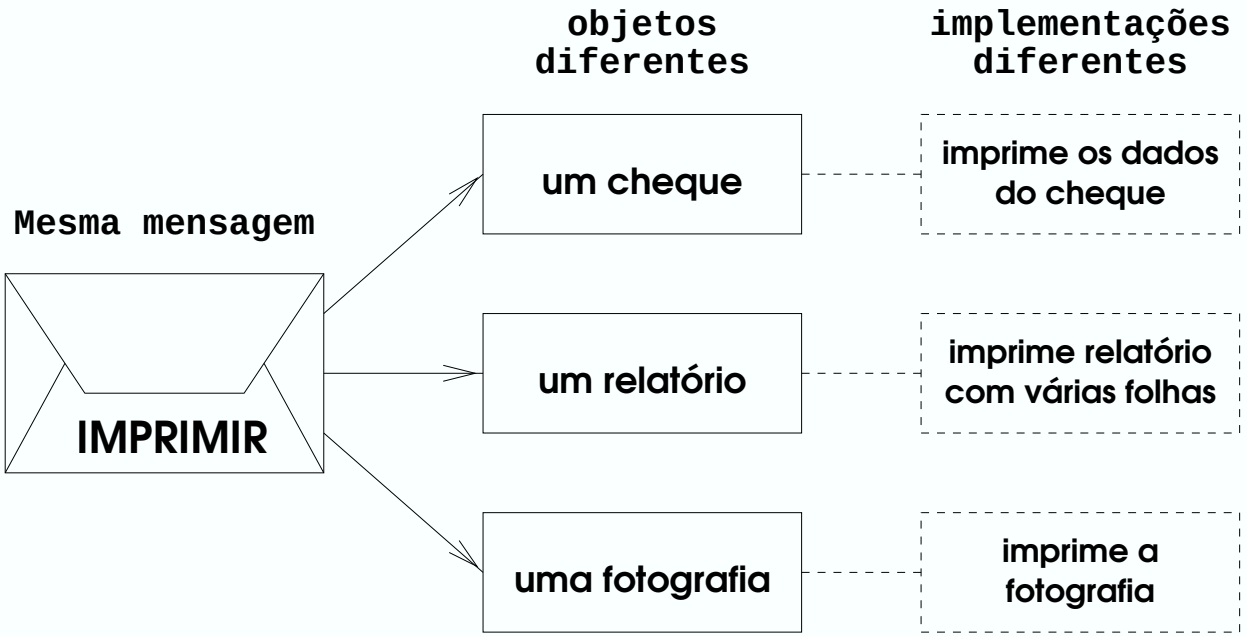
- $LENGTH$: função cujos argumentos são listas
- A : o tipo de elemento armazenado na lista pode ser qualquer um
- NUM : o tipo de retorno é um número inteiro

Exemplo de uma Linguagem Polimórfica (II)

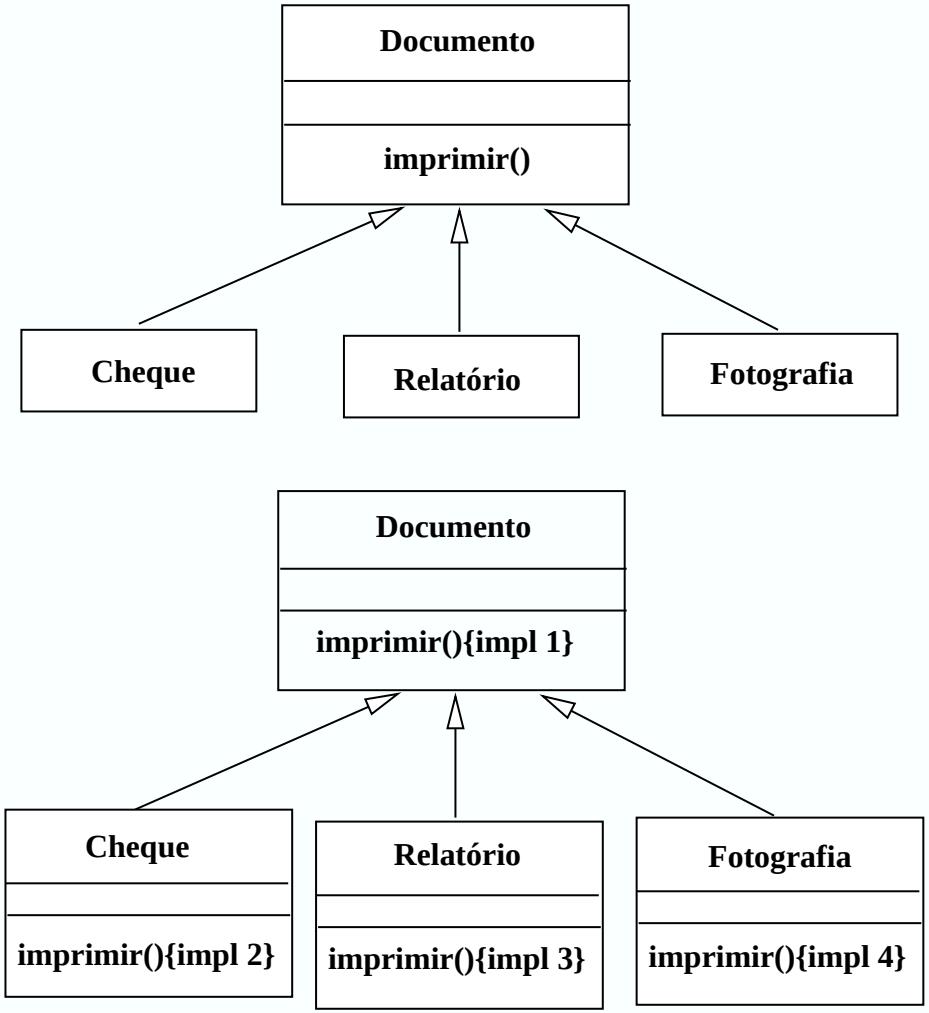
- Uma linguagem monomórfica força o programador a definir diferentes funções para retornar o comprimento de uma lista de inteiros, de reais, etc.. Pascal e C são exemplos de linguagens monomórficas
- Uma linguagem monomórfica exige que os tipos dos parâmetros sejam especificados na definição da função
- Não é possível definir uma função verdadeiramente polimórfica onde o tipo de um parâmetro permanece indefinido até o momento da execução do programa.

Polimorfismo no Modelo de Objetos

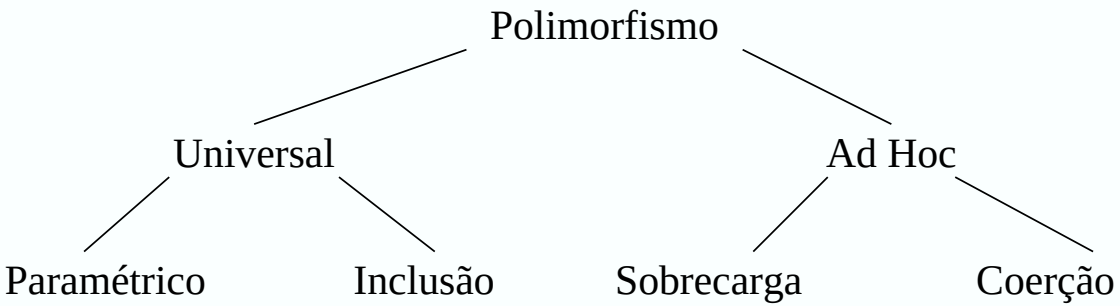
Polimorfismo significa que diferentes tipos de objetos podem responder a uma mesma mensagem de diferentes maneiras



Hierarquia de Tipos e Subtipos



Classificação de Polimorfismo (I)



Classificação de Polimorfismo (II)

- O polimorfismo é dito **universal** ou **verdadeiro** quando uma função ou tipo trabalha uniformemente para uma gama de tipos definidos na linguagem
- O polimorfismo é dito **ad hoc** ou **aparente** quando uma função ou tipo parece trabalhar para alguns tipos diferentes e pode se comportar de formas diferentes para cada tipo. Como exemplo, podemos mencionar o "write" do Pascal
- O conceito de "polimorfismo" está associado com reutilização

Coerção (I)

- Forma limitada de polimorfismo. A linguagem tem um mapeamento interno entre tipos. Se num contexto particular o tipo requerido é diferente do tipo dado, a linguagem verifica se existe uma coerção, i.e., uma conversão de tipos interna.
  Exemplo:

```
procedure soma(a:real; b:real){...}
   ...
   var c:integer; d:real;
   soma(c,d); // o valor inteiro é convertido para real
```

Coerção (II)

- Se não houvesse coerção, teriam que ser definidos diversos procedimentos:

```
procedure soma1(a:real; b:real){...};
procedure soma2(a:integer; b:real){...};
procedure soma3(a:real; b:integer){...};
procedure soma4(a:integer; b:integer){...};
```

- Note que todos os procedimentos têm nomes diferentes, mas suas implementações internas são iguais
- Portanto, a coerção proporciona reutilização de código

Exemplos de Coerção em Java (I)

- Em Java são executadas as seguintes conversões implicitamente:

```
byte to short, int, long or double
short to int, long, float or double
char to int, long, float ou double
int to long, float or double
long to float or double
float to double
```

Exemplos de Coerção em Java (II)

- Todas as conversões implícitas realizadas em Java são consideradas promoções de tipo, isto é, o valor inicial é um tipo cujo domínio está contido no domínio do tipo resultante. Não pode haver truncamento no resultado
- Conversão entre inteiros (short, int ou long) e entre reais (float ou double) pode resultar em perda de precisão

Sobrecarga (I)

- Permite que um "nome de função" possa ser usado mais de uma vez com diferentes tipos de parâmetros. Exemplo:

```
procedure soma(a:integer; b:integer){
   // soma dois inteiros}
procedure soma(a:string; b:string){
   // concatena duas strings}
```

Sobrecarga (II)

- Note que os nomes dos dois procedimentos são iguais, mas as suas semânticas e implementações internas são diferentes
- A noção de sobrecarga permite a reutilização de um nome

Exemplo de Sobrecarga (I)

- Sobrecarga de métodos construtores:

```
public ContaCor(){...} // construtor ‘‘default’’
public ContaCor(String nome, float val, int num,
                int pwd){...}
```

- Sobrecarga de operadores: um operador da linguagem pode ter diferentes significados, dependendo do tipo do parâmetro aplicado.
  Exemplo: $a+=b$
  Significado(1): “adicione o valor b ao atributo a”
  Significado(2): “inclua o elemento b no conjunto a”

Exemplos de Sobrecarga (II)

- Java não permite sobrecarga de operadores, apenas de métodos
- C++ permite sobrecarga de operadores e de métodos

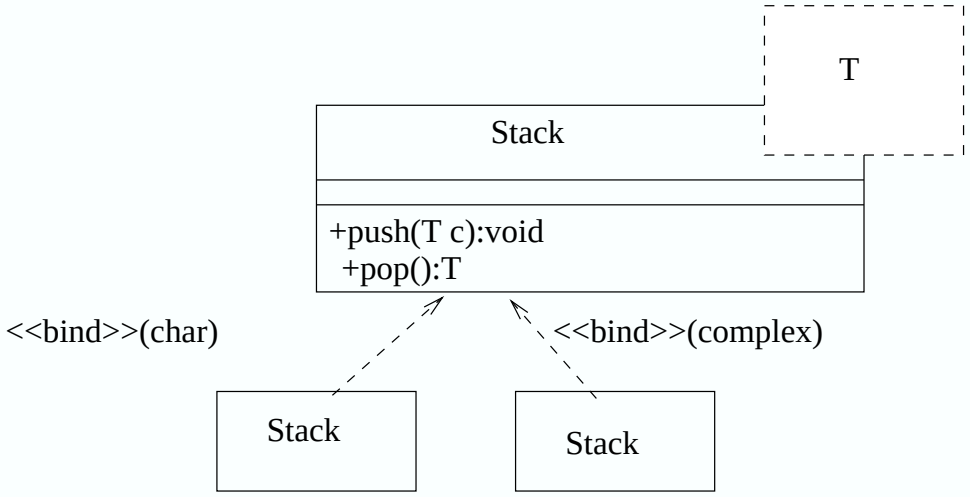
Polimorfismo Ad Hoc X Polimorfismo Universal

- No polimorfismo **"ad hoc"** não existe um modo único e sistemático de determinar o tipo de resultado de uma função em termos dos tipos dos seus argumentos
- O polimorfismo **"ad hoc"** trabalha com um número limitado de tipos de um modo não sistemático, enquanto que o polimorfismo **universal** trabalha potencialmente com um conjunto infinito de tipos de modo disciplinado

Polimorfismo Paramétrico

- Uma única função (ou tipo) é codificada e ela trabalhará uniformemente num intervalo de tipos
- Exemplo de tipo paramétrico $Stack(T)$, onde T é o tipo do elemento a ser empilhado.
  Dessa forma, um programa "genérico" sobre pilhas pode ser escrito independentemente do tipo dos elementos que serão empilhados

Polimorfismo Paramétrico em UML



(a) ligação explícita

Stack<char>

Stack<complex>

(b) ligação implícita

Tipo Paramétrico em C++ (I)

```
template<class T>
class Stack{
   private:
      T* s;
      int top;
      ...
   public:
      Stack(){s=new T[1000]; top = -1;}
      void push(T c){s[++top] = c;}
      T pop(){return (s[top]);}
}
```

Tipo Paramétrico em C++ (II)

- Uso do tipo stack:

```
Stack<char> stk_char; //pilha de 1000 elem. do tipo char
Stack<complex> stk_complex; //pilha de 1000 elementos
                            //do tipo complex
```

- Neste exemplo, dois tipos novos foram gerados, Stack$< char >$ e Stack$< complex >$, a partir do molde Stack$< T >$

Polimorfismo Paramétrico em C++ (I)

- Procedimento polimórfico que troca os valores de duas variáveis de tipos iguais:

```
template<class T>
void troca(T& a, T& b){
   T temp = a;
   a = b;
   b = temp;
} // fim da troca
```

Polimorfismo Paramétrico em C++ (II)

- Procedimento polimórfico que copia o vetor b no vetor a:

```
template<class T>
void copy(T a[], T b[], int n){
   for(int i=0; i < n; ++i){
      a[i] = b[i];
   }
} // fim de copy
```

Polimorfismo Paramétrico em C++ (III)

- Uso do procedimento copy:

```
double f1[50], f2[50];
copy(f1, f2, 50);
char c1[25], c2[50];
copy(c1, c2, 10);
```

Polimorfismo Paramétrico em Java (I)

- A partir da versão 1.5.0 lançada em 2005, a linguagem Java incorporou o conceito de tipo genérico ("generics")
- O sistema que copia o conteúdo de dois vetores genéricos, que foi escrito anteriormente em C++, poderia ser codificado da seguinte forma:

```
public class Copier<E>{
   public void copy(E a[], E b[], int n){
      for (int i = 0; i < n; ++i){
         a[i] = b[i];
      }
   }
```

Polimorfismo Paramétrico em Java (II)

- Uso do método copy:

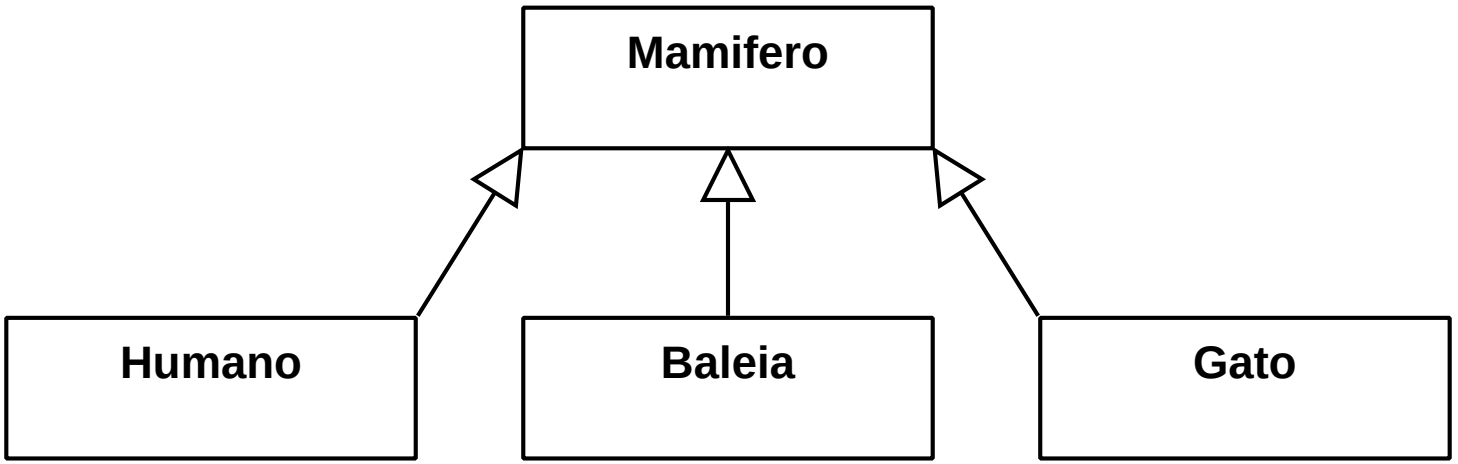
```
public void main(){
   double f1[] = new double[50];
   double f2[] = new double[50];
   Copier cpDouble = new Copier<double>();
   cpDouble.copy(f1, f2, 50);

   char c1[] = new char[25];
   char c2[] = new char[25];
   Copier cpChar = new Copier<char>();
   cpChar.copy(c1, c2, 10);
   }
}
```

Polimorfismo de Inclusão (I)

- Polimorfismo de **inclusão** é o estilo de polimorfismo encontrado em todas as linguagens orientadas a objetos
- O polimorfismo de inclusão está relacionado com a existência da hierarquia de generalização/especialização e com o conceito de subtipo
- Definição de **subtipo**: um tipo S é um subtipo de T se e somente se S proporciona pelo menos o comportamento de T
- A noção de subtipo implica que elementos do subconjunto também pertencem ao superconjunto

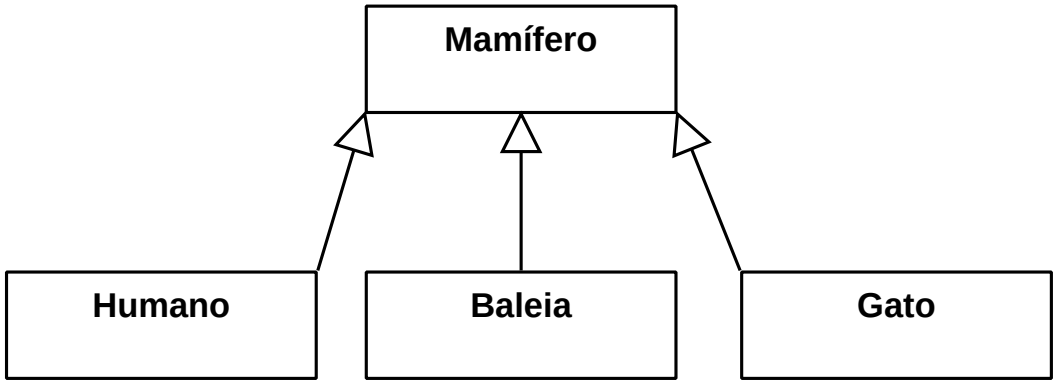
Polimorfismo de Inclusão (II)

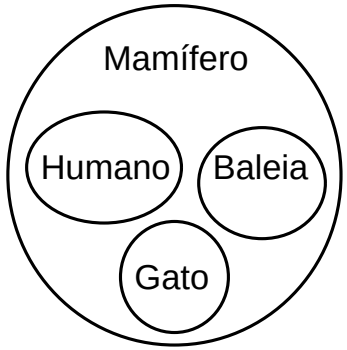


Polimorfismo de Inclusão (III)

- Considere um tipo representado por **Baleia**, subtipo de um tipo mais geral chamado **Mamífero**
- Nesse caso, todo objeto de um subtipo pode ser usado no contexto do supertipo no sentido de que toda baleia é um mamífero e pode ser operado por todas as operações que são aplicadas ao tipo **Mamífero**

Exemplo 1: Polimorfismo de Inclusão (I)





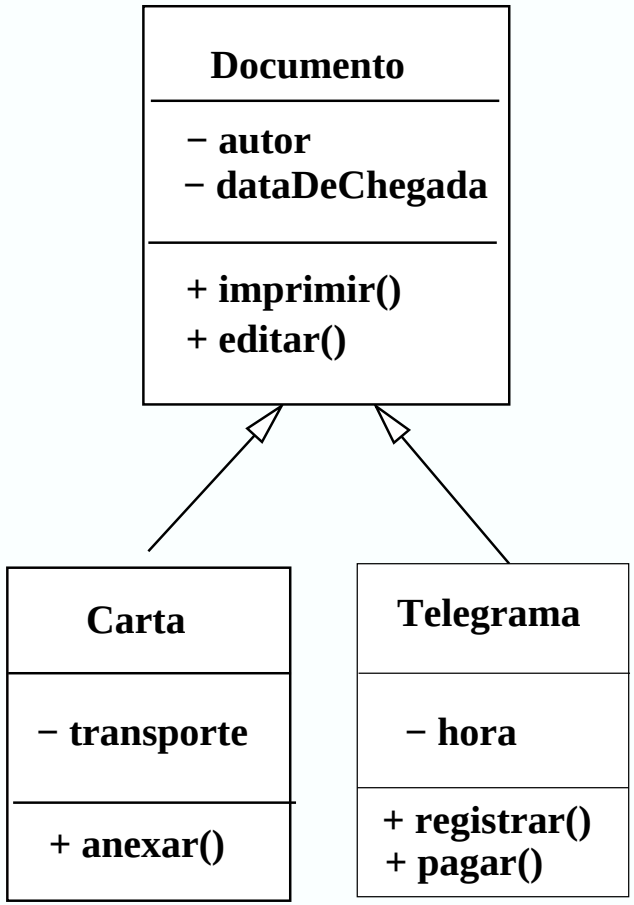
Exemplo 1: Polimorfismo de Inclusão (II)

- A definição da hierarquia de generalização/especialização favorece o uso do polimorfismo de inclusão pois as subclasses (ou subtipos) herdam automaticamente todas as operações da superclasse (ou supertipo)
- Com a construção da hierarquia, todas as operações do tipo **Mamifero** são capazes de operar tanto sobre objetos do tipo **Mamifero**, quanto dos seus subtipos: **Humano**, **Gato** e **Baleia**

Exemplo 1: Polimorfismo de Inclusão (III)

- As operações do tipo **Mamífero** são "reentrantes" nos tipos **Humano**, **Gato** e **Baleia**
- As operações do tipo **Mamífero** que são herdadas pelos subtipos são **polimórficas** (i.e., polimorfismo de inclusão)

Exemplo 2: Polimorfismo de Inclusão (I)
(sem redefinição de operações)



Exemplo 2: Especificação dos Tipos (II) (sem redefinição de operações)

Documento = {imprimir(), editar()}

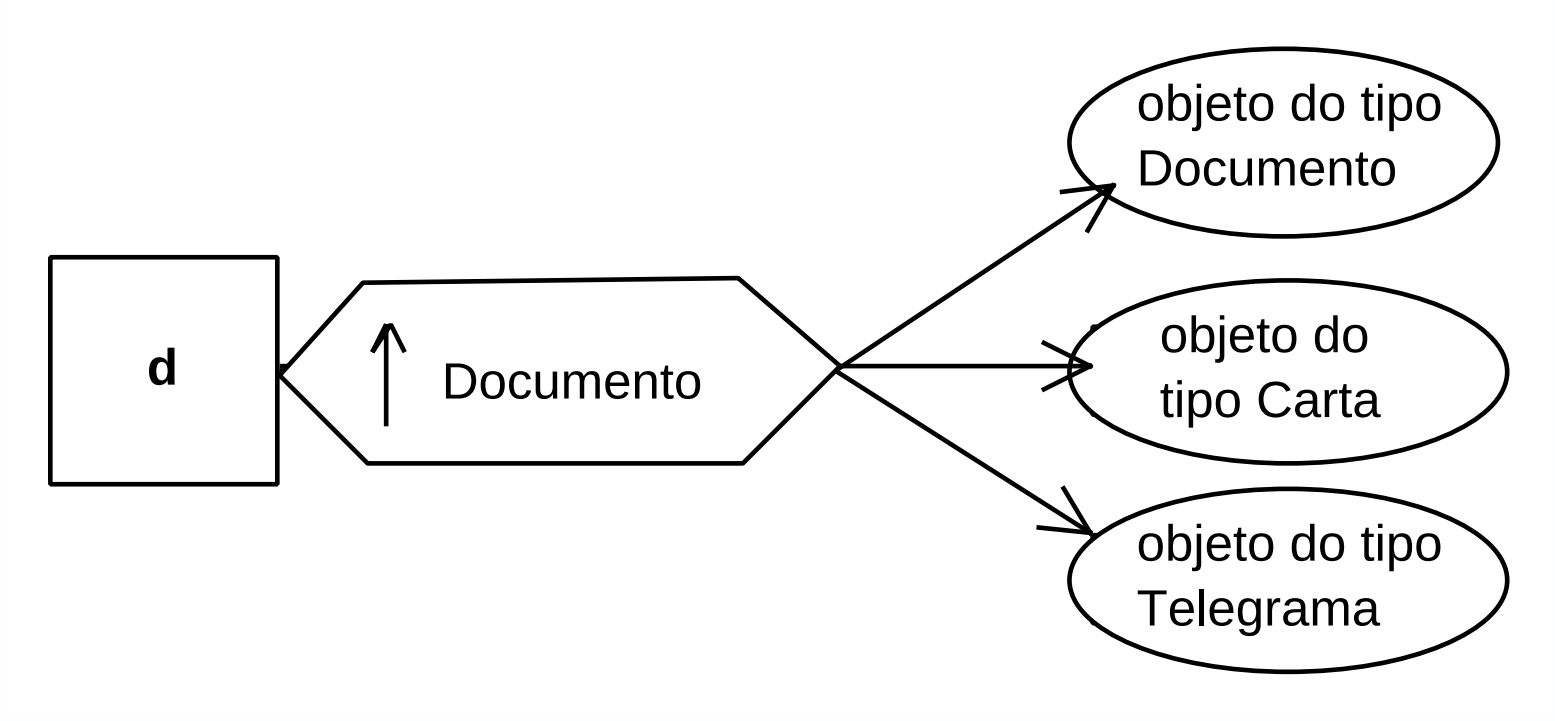
Carta = {imprimir(), editar(), anexar()}

Telegrama = {imprimir(), editar(), registrar(), pagar()}

Exemplo 2: Polimorfismo de Inclusão (III) (sem redefinição de operações)

```
Documento d = new Documento();
d.imprimir(); //Documento
d = new Carta();
d.imprimir(); // Documento
d.anexar(); // erro
d = new Telegrama();
d.imprimir(); // Documento
d.registrar(); // erro
d.pagar(); // erro
```

Exemplo 2: Polimorfismo de Inclusão (IV) (sem redefinição de operações)



Exemplo 2: Polimorfismo de Inclusão (V)

- A variável *d* é do tipo ↑Documento, e, portanto, ela aceita apenas as operações definidas pelo tipo `Documento`, ou seja, *imprimir()* e *editar()*
- *d* pode referenciar objetos do tipo `Documento` e também objetos que são subtipos de `Documento`. Isto é possível porque as operações *imprimir()* e *editar()* também pertencem aos tipos `Carta` e `Telegrama` (foram herdadas pelos subtipos)
- Quando as operações *imprimir()* e *editar()* são enviadas para *d*, tanto os objetos do tipo `Documento` quanto do tipo `Carta` e `Telegrama` são capazes de respondê-las

Exemplo 2: Polimorfismo de Inclusão (VI)

- Observe que a operação *d.imprimir()* funciona uniformemente para todos os documentos; como se eles fossem de um mesmo tipo, independentemente se eles sejam cartas ou telegramas
- O polimorfismo de inclusão foi batizado com esse nome porque as operações polimórficas de um tipo estão "incluídas" nos seus subtipos
- A classe `Object` é a raiz de qualquer classe criada em Java
- Os métodos da classe `Object` são exemplos de polimorfismo de inclusão, pois eles são capazes de operar uniformemente sobre objetos de qualquer tipo em Java

As Diferentes Categorias de Polimorfismo

- Em C++ e Java, temos polimorfismo paramétrico ("template" / "tipos genéricos"), sobrecarga, polimorfismo de inclusão e coerção

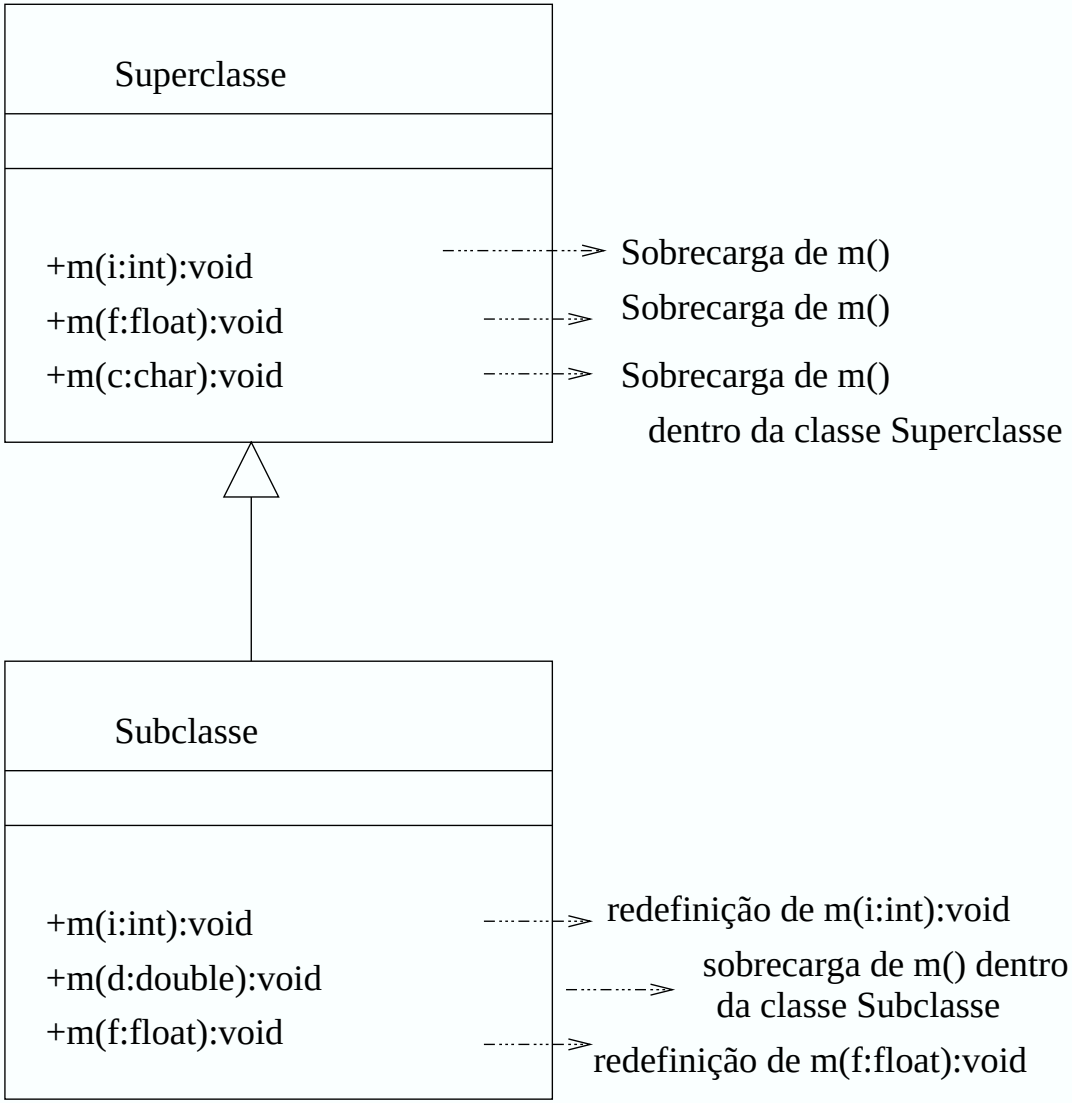
Redefinição de Operações

- É um caso especial de polimorfismo de inclusão, quando uma classe define uma forma de implementação especializada para um método herdado da superclasse.
- Em C++, as funções virtuais numa hierarquia de tipos permitem que uma seleção da implementação mais especializada de um método seja escolhida em tempo de execução.
- Todos os métodos em Java são virtuais por definição, isto é, todos os métodos implementam **acoplamento dinâmico** implicitamente.

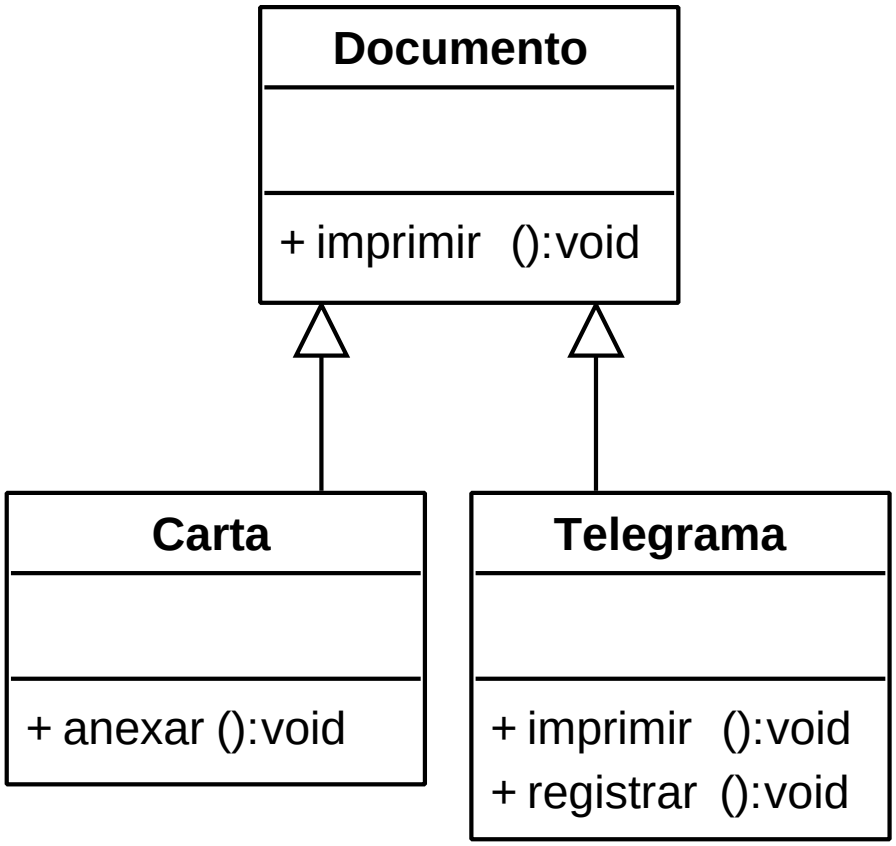
Redefinição X Sobrecarga de Operações (I)

- **Redefinição:** a nova operação deve ter a mesma assinatura da operação herdada, isto é, elas devem ser idênticas quanto ao nome da operação e à lista de parâmetros (mesmo número de parâmetros, com os mesmos tipos e declarados na mesma ordem)
- O tipo do resultado de retorno não faz parte da assinatura da operação e ele não pode ser mudado uma vez definido.
- **Sobrecarga:** ocorre quando existe apenas coincidências nos nomes das operações, isto é, as listas de parâmetros não são idênticas.

Redefinição X Sobrecarga de Operações (II)



Polimorfismo de Inclusão (I)
(com redefinição de operações)



Polimorfismo de Inclusão (II)
(com redefinição de operações)

```
Documento d = new Documento();
d.imprimir(); // imprimir() de Documento
d = new Carta();
d.imprimir(); // imprimir() de Documento
d = new Telegrama();
d.imprimir(); // imprimir() de Telegrama
              //(acoplamento dinâmico)
```

Acoplamento Estático e Dinâmico

- O acoplamento é **estático** ou **adiantado** ("early") se ocorre antes do tempo de execução, e permanece inalterado durante a execução do programa
- O acoplamento é **dinâmico** ou **atrasado** ("late") se ocorre durante o tempo de execução, e muda no curso da execução do programa
- Exemplo: acoplamento de nomes de operações a métodos (isto é, código executável)
- Em POO, combina-se verificação estática de tipos com acoplamento dinâmico

Acoplamento Dinâmico em C++

```
class Documento{
   public:
      virtual void imprimir(); // acoplamento dinâmico
      void editar(); // acoplamento estático
}
class Telegrama : public Documento{
   void imprimir();
   void registrar();
}
...
Documento *d = new Documento();
d -> imprimir(); // Documento :: imprimir()
d = new Telegrama();
d -> imprimir(); // Telegrama :: imprimir()
```

Acoplamento Dinâmico em Java

- Java é considerada uma linguagem OO "pura" que utiliza o conceito de acoplamento dinâmico implicitamente
- Não existe a necessidade do uso explícito de acoplamento dinâmico através da palavra reservada *virtual* como em C++
- Em princípio, todas as operações em Java utilizam acoplamento dinâmico
- O modificador *final* de Java implementa acoplamento estático de métodos
- Uma operação *final* é considerada não polimórfica e, portanto, não pode ser redefinida

Exemplo de Acoplamento Estático (I)

- Se Java implementasse apenas acoplamento estático, todas as chamadas da operação *imprimir()* enviadas para os objetos do tipo Documento seriam substituídas pelo código binário de *imprimir* definido pela classe `Documento` durante o tempo de compilação.
- Esse acoplamento (ligação ou amarração) permaneceria imutável (isto é, estática) durante todo o tempo de execução do programa.

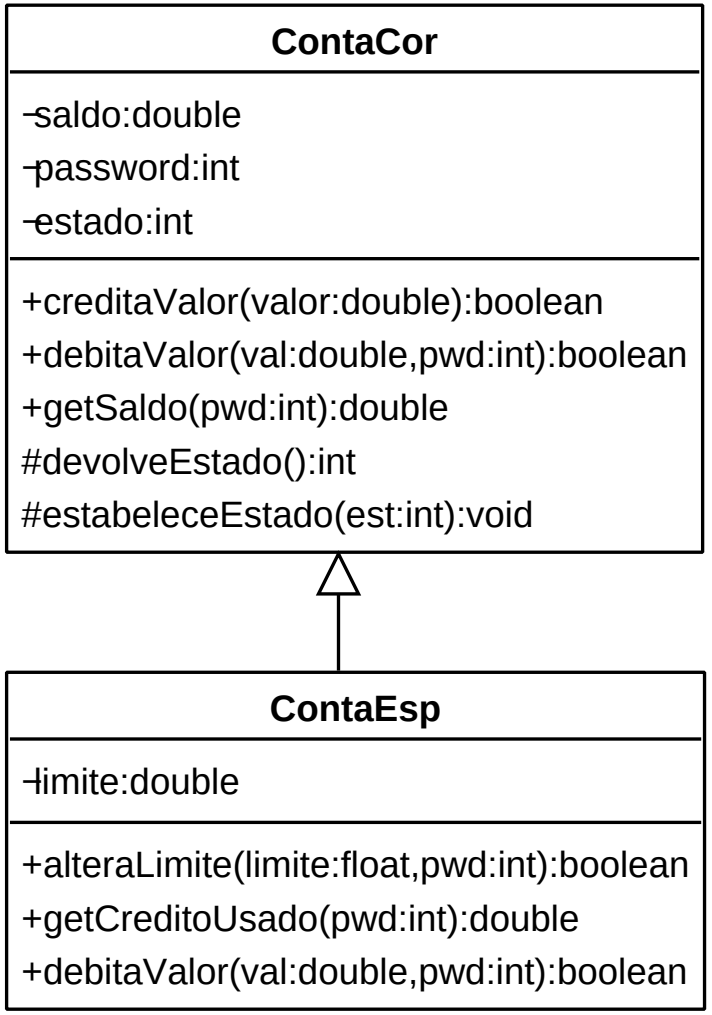
Exemplo de Acoplamento Estático (II)

```
Documento d = new Documento();
d.imprimir(); // ativa imprimir() de Documento
d = new Carta();
d.imprimir(); // ativa imprimir() de Documento
d = new Telegrama();
d.imprimir(); // ativa imprimir() de Documento
              // e não de Telegrama, pois o acoplamento
              // é estático e não dinâmico
Telegrama t = new Telegrama();
t.imprimir(); // ativa imprimir() de Telegrama
```

Exemplo `ContaCor` **(I)**

- Uma conta corrente deve se tornar inativa se durante sua movimentação o seu saldo se igualar a zero, não podendo mais realizar saques ou consultar o saldo
- Uma conta especial, que tenha um limite maior que zero, pode ter o seu saldo igualado a zero sem que a conta se torne inativa
- Uma conta especial com limite igual a zero deve ser tratada como uma conta comum

Exemplo `ContaCor` (II)



Exemplo `ContaCor` (III)

- No exemplo, o saldo não fica negativo (nem para uma conta comum, e nem para uma conta especial).
- A regra original que determina a desativação da conta com saldo zero está implantada no método *debitaValor()* da classe `ContaCor`.
- Portanto, esse método deve ser redefinido na classe `ContaEsp`.

Exemplo ContaCor (IV)

```
ContaCor c1; // c1 pode referenciar objetos do tipo
             // ContaCor e também seus subtipos
c1 = new ContaCor(...);
c1.creditaValor(...); // OK
c1.debitaValor(...); // OK, (implementação de ContaCor)
c1.getSaldo(...); // OK
c1 = new ContaEsp(...);
c1.creditaValor(...); // OK
c1.debitaValor(...); // OK, (implementação de ContaEsp)
c1.getSaldo(...); // OK
c1.alteraLimite(...); // ERRO, c1 é do tipo ContaCor
c1.getCreditoUsado(...); // ERRO, c1 é do tipo ContaCor
```

Exemplo `ContaCor` (V)

```
ContaEsp c2 = new ContaEsp();
c2.creditaValor(...); // OK
c2.debitaValor(...); // OK (implementação de ContaEsp)
c2.getSaldo(...); // OK
c2.alteraLimite(...); // OK, c2 é do tipo ContaEsp
c2.getCreditoUsado(...); // OK, c2 é do tipo ContaEsps
```

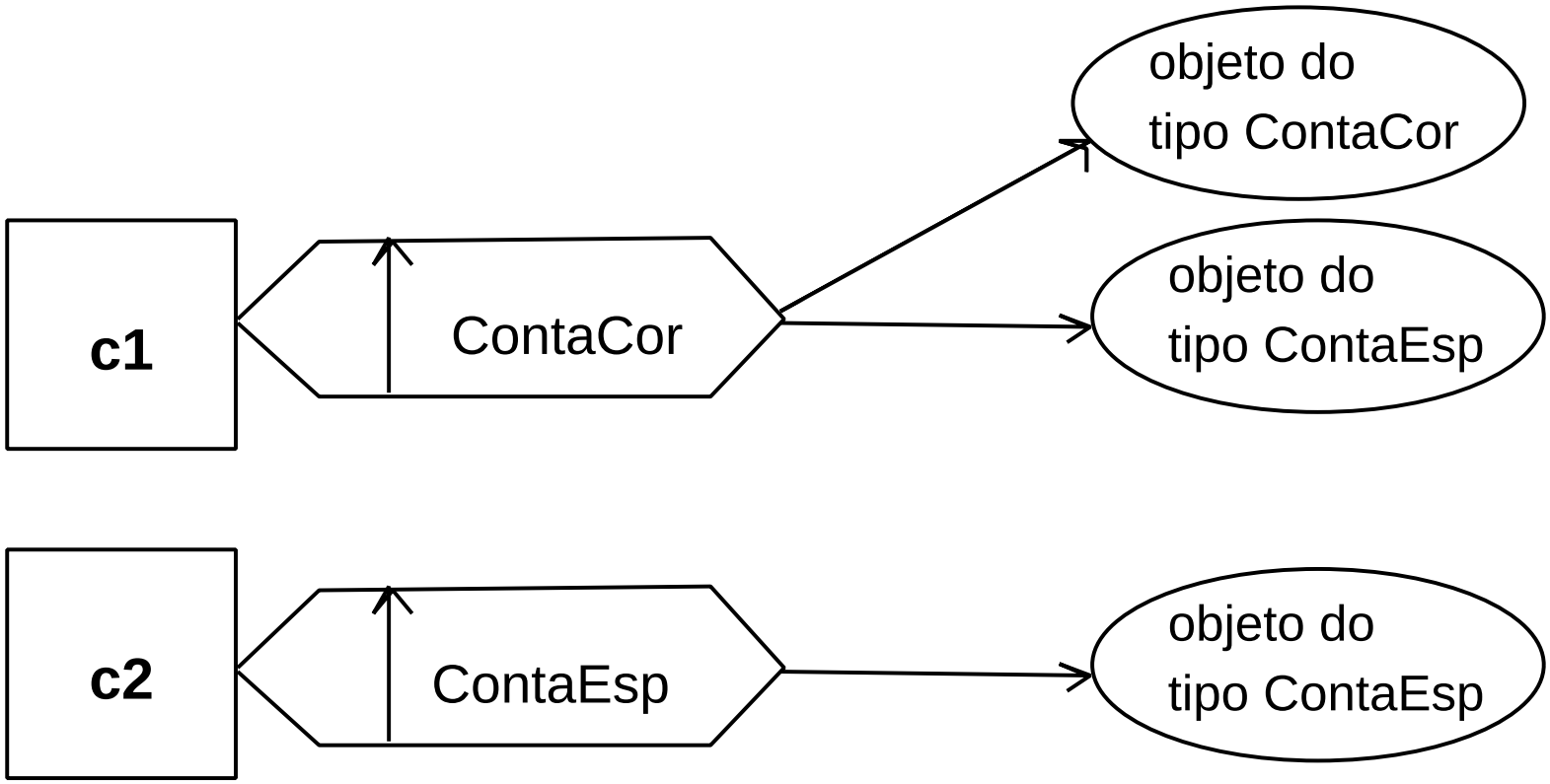
- **Lembrete:**
  tipo `ContaCor` = {creditaValor(...), debitaValor(...), getSaldo(...)}
  tipo `ContaEsp` = {creditaValor(...), debitaValor(...), getSaldo(...), alteraLimite(...), getCreditoUsado(...)}

Exemplo `ContaCor` (VI)

ContaCor c1 = new ContaCor();
ContaEsp c2 = new ContaEsp();
c1 = c2; // atribuição válida

- A atribuição $c1 = c2$ acopla dinamicamente a variável *c1* do tipo `ContaCor` a um objeto de um tipo diferente (isto é, tipo `ContaEsp`) do seu tipo estático.
- Em princípio, seria uma violação de tipo atribuir um objeto de um tipo diferente à variável *c1* que, de acordo com a sua especificação de tipo, deve referenciar apenas objetos do tipo `ContaCor`.

Exemplo `ContaCor` (VII)



Exemplo ContaCor (VIII)

Se `ContaEsp` é um subtipo de `ContaCor`, a atribuição é válida:

```
ContaCor c1 = new ContaCor();
ContaEsp c2 = new ContaEsp();
c1.debitaValor(...); // debitaValor(...) de ContaCor
c1 = c2;
c1.debitaValor(...); // debitaValor(...) de ContaEsp
```

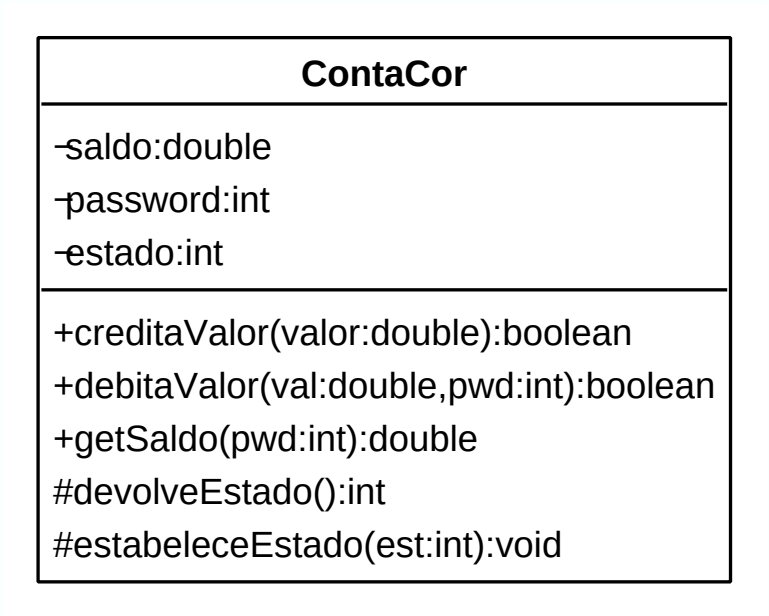
Exemplo `ContaCor` (IX)

- Em C++, o programador deve pedir explicitamente o acoplamento dinâmico para uma operação.
- Em C++, a operação deve ser declarada *virtual* na classe base e ser redefinida na classe derivada.
- Em Java, a operação da classe base deve ser apenas redefinida na classe derivada.
- **Ponto Crucial:** todas as operações redefinidas na classe derivada têm a responsabilidade de manter a mesma semântica dos serviços oferecidos pela classe base.

Exemplo `ContaCor` (X)

```
public boolean debitaValor(double val, int pwd){
  boolean r;
  r = super.debitaValor(val, pwd);
  if((limite > 0) && (devolveEstado()==2)){ // conta
                                    //especial inativa
     estabeleceEstado(1); // conta especial ativa
  }
  return r;
}
```

Exemplo `ContaCor` (XI)



Exemplo `ContaCor` (XII)

```
class ContaCor{
...

   protected void estabeleceEstado(int e){estado = e;}
   protected int devolveEstado(){return estado;}
}
```

Exemplo `ContaCor` (XIII)

- Note que são utilizados dois novos métodos (*devolveEstado()* e *estabeleceEstado()*) que inspecionam o valor do atributo privado `estado` definido na classe `ContaCor`.
- Esses dois métodos devem ser incluídos na classe `ContaCor` com visibilidade protegida para que a classe `ContaEsp` possa chamá-los.

Polimorfismo e Verif. Estática de Tipos (I)

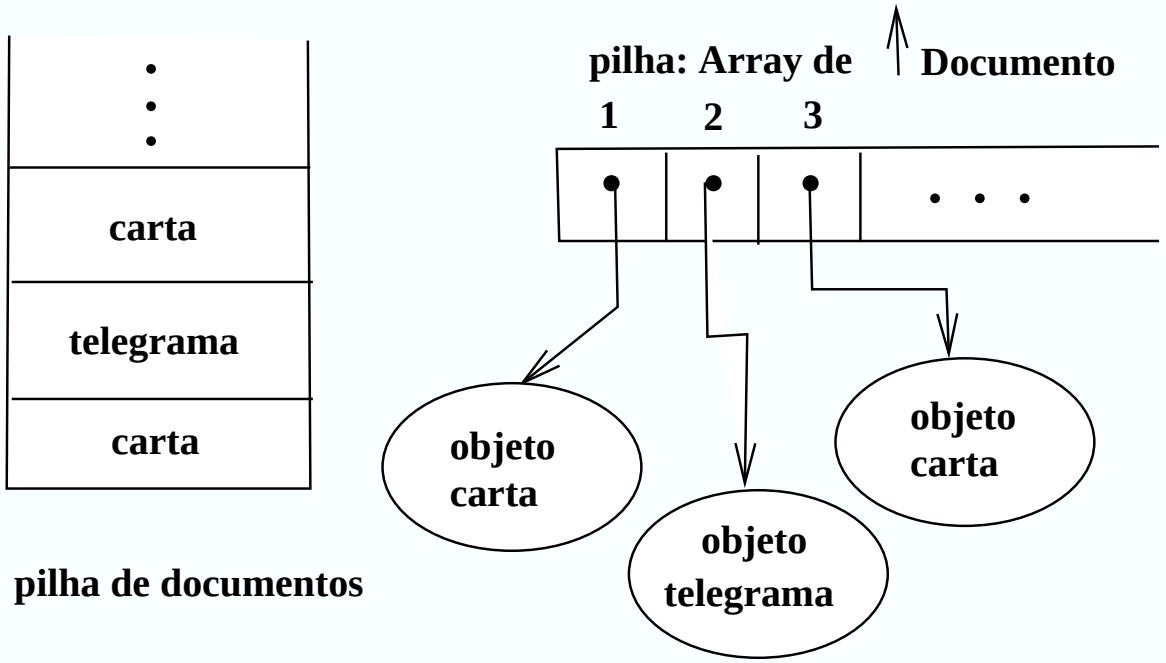
- Polimorfismo existe em linguagens tipadas e não-tipadas.
- O conceito de tipo permite que o compilador verifique estaticamente que todas as mensagens enviadas para um objeto através de uma referência serão entendidas e executadas em tempo de execução.
- Em linguagens estaticamente tipadas, mensagens serão sempre entendidas pelo objeto recipiente devido à tipagem forte.

Polimorfismo e Verif. Estática de Tipos (II)

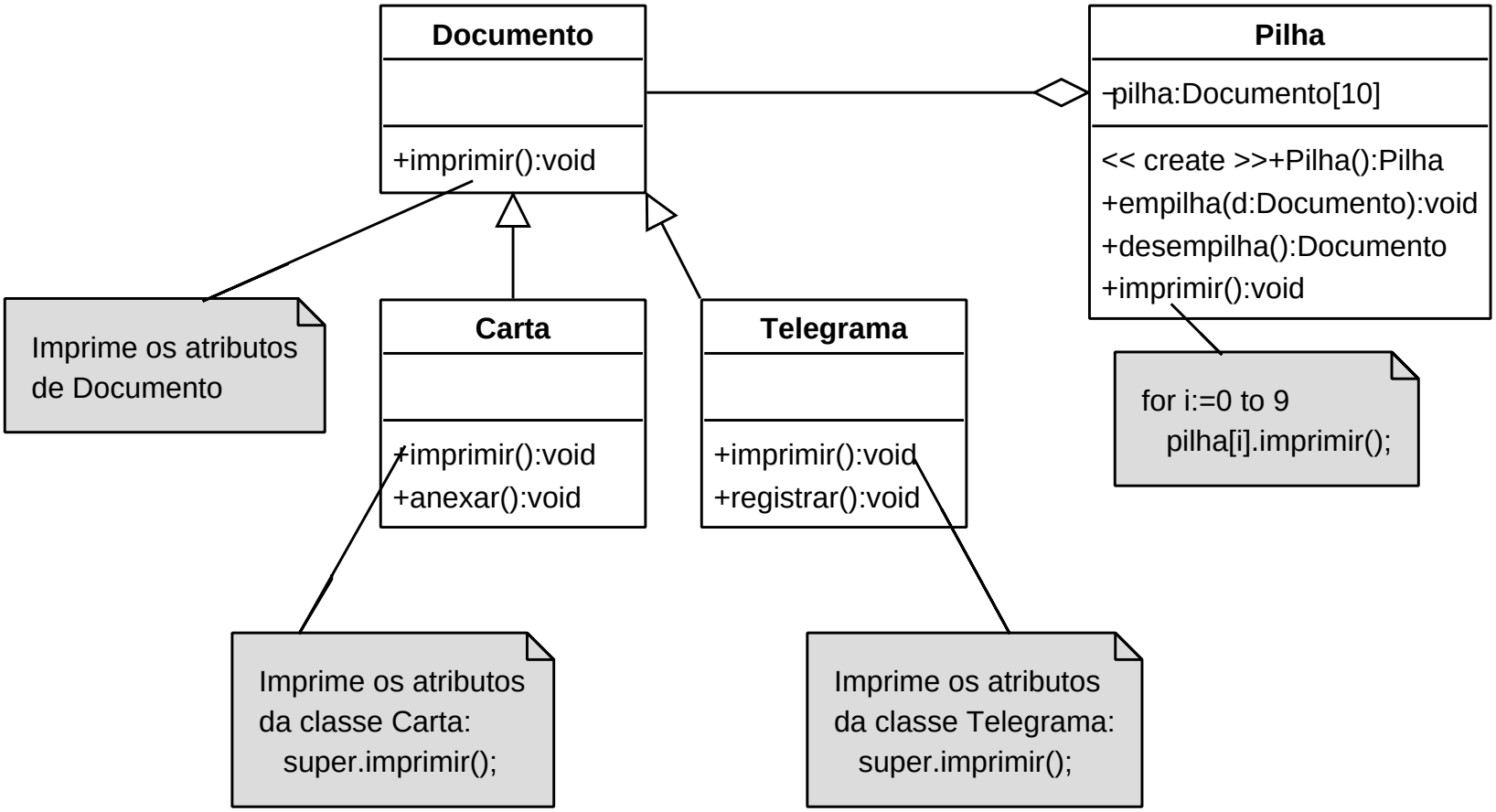
- Em linguagens não-tipadas ou dinamicamente tipadas, qualquer mensagem pode ser enviada para qualquer referência de objeto
- Em SmalltalK ou Python, por exemplo, uma mensagem pode não ser entendida pelo objeto recipiente e, portanto, o objeto remetente deve estar preparado para esta eventualidade
- Isso, entretanto, não interfere no acoplamento dinâmico, que é responsável por encontrar o código mais especializado para um determinado serviço

Pilha Polimórfica de Documentos (I)

- Considere uma Pilha de documentos (cartas, telegramas, etc.) e que você gostaria de imprimí-los.



Pilha Polimórfica de Documentos (II)



Implementação Java (I)

- Numa linguagem OO pode-se produzir o seguinte código:

```
class Pilha{
   private Documento pilha[] = new Documento[10];
   ...
   public void imprimir(){
      for(int i = 0; i < 10; i++){
         pilha[i].imprimir();
      } // fim do for
   } // fim do método imprimir()
}
```

Implementação Java (II)

```
class Documento{
   ...
   public void imprimir(){
    // imprimir todos os atributos
   }
}

class Carta extends Documento{
   ...
   public void imprimir(){
     super.imprimir(); // imprime atributos do pai
    // imprimir todos os atributos específicos
   }
}
```

Implementação Java (III)

```
class Telegrama extends Documento{
   ...
   public void imprimir(){
     super.imprimir(); // imprime atributos do pai
    // imprimir todos os atributos específicos
   }
}

class Pilha{
   private Documento pilha[] = new Documento[10];
   ...
   public void imprimir(){
      for(int i = 0; i < 10; i++){ pilha[i].imprimir(); }
   } // fim do método imprimir()
}
```

Implementação Estruturada (I)

```
program PilhaDocumentos;
type classeDeDocumento = (Carta, Telegrama);
   Documento = record
                  autor: string[40];
                  dataDeChegada: tipoData;
                  case classe: classeDeDocumento of
                  Telegrama: (hora: tipoHora)
                  Carta: (transp: tipoTransporte)
                  end;
   procedure imprimirCarta() begin ... end;
   prodedure imprimirTelegrama() begin ... end;
```

Implementação Estruturada (II)

```
var d1, d2: Documento;
   pilha: array[1..10] of Documento;
   i : integer;
begin
  pilha[1].classe := Carta; // atributo selecionador
  pilha[2].classe := Telegrama; // atributo selecionador
  i := 1;
  while (not vazia(pilha)) do begin
     case pilha[i].classe of
        Carta: imprimirCarta();
        Telegrama: imprimirTelegrama();
     end; // fim do case
  end; // fim do while
end. // fim do programa
```