

Modularização em LOOs

- Estrutura de programas definida por **interfaces** entre classes
- Interface entre classes A e B especificada por conjunto de nomes de A que podem ser usados em B e vice-versa
- Modificação/extensão mais fácil de programas:
 - ★ divisão em classes com interfaces claras e bem definidas
 - ★ e com **encapsulamento** de dados: mudança na implementação não acarreta mudança em outras classes



Facilidade de modificação

Mecanismos para facilitar reuso com possibilidade de extensão e especialização:

- **Herança**: aproveitamento de definições, com possibilidade de redefinição
- **Subtipagem**: definições existentes podem operar com instâncias das novas definições
- **Associação dinâmica**: redefinições automaticamente usadas se objeto é instância da redefinição; permite que análise de casos se restrinja a criação de objetos.

Controle de visibilidade de nomes

Acesso permitido	<code>private</code>	sem atributo	<code>protected</code>	<code>public</code>
mesma classe	Sim	Sim	Sim	Sim
mesmo pacote, outra classe	Não	Sim	Sim	Sim
outro pacote, subclasse	Não	Não	Sim	Sim
outro pacote, fora de subclasse	Não	Não	Não	Sim

Controle de redefinição em subclasse

- Atributo `final` em definição de método indica que método não pode ser redefinido em subclasse.
- Garante que comportamento de um método não vai ser modificado em nenhuma subclasse desse método. Ex: método para verificação da validade de senha secreta.
- Atributo `final` em declaração de classe especifica que nenhuma subclasse dessa classe pode ser definida.



Classes e métodos abstratos

- Classe **abstrata** — declarada com atributo `abstract` — contém um ou mais métodos abstratos
- Método abstrato — declarado com atributo `abstract` — não é implementado: apenas assinatura especificada
- Implementação de método abstrato deve ser feita em subclasse não abstrata
- Classe abstrata provê comportamento de alguns métodos e especifica apenas interface de outros

Classes e métodos abstratos: Exemplo

```
abstract class Benchmark
{
  abstract void benchmark();
  public long repita(int n)
  {
    long inicio = System.currentTimeMillis();
    for (int i=0; i<n; i++) benchmark();
    return (System.currentTimeMillis() - inicio);
  }
}
```

Interfaces

- **Interface** contém assinaturas de métodos e constantes.
- Classes podem implementar uma ou mais interfaces, definindo implementações para métodos cujas assinaturas foram especificadas nessas interfaces.

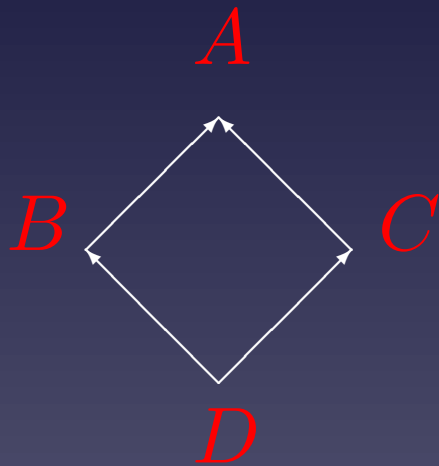
```
interface Ponto
{ float x(); float y();
  void move(float dx, float dy);
  float distancia(Ponto p); }
```

```
public class PCart implements Ponto
{
    private float x, y;
    public float x() { return x; }
    public float y() { return y; }
    public PCart(float a, float b) {x=a; y=b;}
    public void move (float dx, float dy)
    { x+=dx; y+=dy; }
    public float distância (Ponto p)
    { float dx = x - p.x(), dy = y - p.y();
      return (float) Math.sqrt(dx*dx+dy*dy); }
}
```

Herança de interfaces

```
interface CoresBasicas
{ int vermelho=1, verde=2, azul=3; }
interface CoresDoArcoIris extends CoresBasicas
{ int amarelo=4, laranja=5, anil=6, violeta=7; }
interface Cores extends CoresDoArcoIris
{ int carmim=8, ocre=9, branco=0; preto=10; }
interface PontoColorido extends Cores, Ponto
{ Cores cor(); void mudaCor(Cores cor); }
```

Porque Java não provê suporte a herança múltipla de classes



Variável declarada/usada em classe herdada por caminhos distintos

Considere a tal variável: uso de expressão $e.a$ em D — onde e tem tipo A — ambíguo.

Ambigüidade no uso de constantes de mesmo nome em interfaces distintas

```
interface A { int x = 1; }
```

```
interface B { float x = 2.0f; }
```

```
class C implements A, B  
{ public static void main(String[] a)  
  { System.out.println(x); }  
}
```

Uso de constantes de mesmo nome em interfaces distintas

```
interface A { int x = 1; }
```

```
interface B { float x = 2.0f; }
```

```
class C implements A, B  
{ public static void main(String[] a)  
  { System.out.println(A.x + B.x); }  
}
```

Subtipagem

Regra de subtipagem:

Se expressão tem tipo T ,
tem também qualquer supertipo de T

Implica que em qualquer ponto de um programa em que é permitido usar expressão de tipo T podemos usar expressão de subtipo de T

(pois expressão de subtipo de T pode ser considerada como expressão de tipo T)

Subtipagem: Exemplo

```
class A
{ int a = 1; void m1 { a++; } }

class B extends A
{ int b = 2;
  void m2()
  { A a = new A(); B b = new B();
    a = b; a.m1(); }
}
```

Associação dinâmica

- Método a ser executado — em chamada “*e.m()*” — determinado (dinamicamente) conforme tipo do objeto denotado por *e*
- Associação dinâmica evita *análises de casos*: testes e chamada para cada tipo possível para o objeto
- Conseqüência: programa mais claro, conciso, e mais fácil de ser modificado

```
class A
{ int a = 1;
  void m() { System.out.println(a); } }
class B extends A
{ int a = 2;
  void m() { System.out.println(a); } }
class Main
{ public static void main(String[] args)
  { A a = new A(); B b = new B();
    a.m(); a = b; a.m(); } }
```



Invariância

- **Invariância** na redefinição de um método em uma sub-classe: método da subclasse e método de mesmo nome na superclasse têm a mesma assinatura.
- Em Java só é realizada associação dinâmica de nomes a métodos no caso em que há invariância na redefinição.

```
class A
{ int a=1;
  void m() { System.out.println(a); } }
class B extends A
{ int a=2;
  void m(int p) { System.out.println(a+p); } }
class Main
{ public static void main(String[] args)
  { A a = new A(); B b = new B();
    a.m(); a = b;
    a.m(1); // erro! } }
```

Covariância e Contravariância

- **Covariância:** Tipo de parâmetro ou do resultado de redefinição de método em subclasse é *subtipo* do tipo na definição original (e redefinição tem o mesmo número de parâmetros).
- **Contravariância:** Tipo de parâmetro ou do resultado de redefinição de método em subclasse é *supertipo* do tipo original.
- Co e contra-variância não provocam associação dinâmica em Java.

Covariância: considere *motorista.m(v)*

```
class Veiculo
{ String placa; ... }
class Caminhao extends Veiculo
{ int cargaMaxima; ... }
class Motorista
{ void m(Veiculo v) { ...v.placa ...} }
class MotoristaDeCaminhao extends Motorista
{ void m(Caminhao c) { ...c.cargaMaxima ...} } }
```

Associação dinâmica: Exemplo

```
import java.util.Arrays;
class Exemplo_compareTo
{ public static void main (String[] a)
  { Candidato[] cands =
    new Candidato [Integer.parseInt(a[0])];
    /* Cria objetos da classe Candidato e
       armazena-os em cands */
    imprime(cands);
    Arrays.sort(cands);
    imprime(cands); } ... }
```



```
class Candidato implements Comparable
{ String nome;
  int inscricao;

  public int compareTo (Object cand)
  { return nome.compareTo
          ((Candidato) cand.nome); }
}
```