9 – Classes Abstractas e Interfaces

Classe Abstracta

– Classe em que pelo menos um dos métodos de instância não é implementado.

Exemplo:

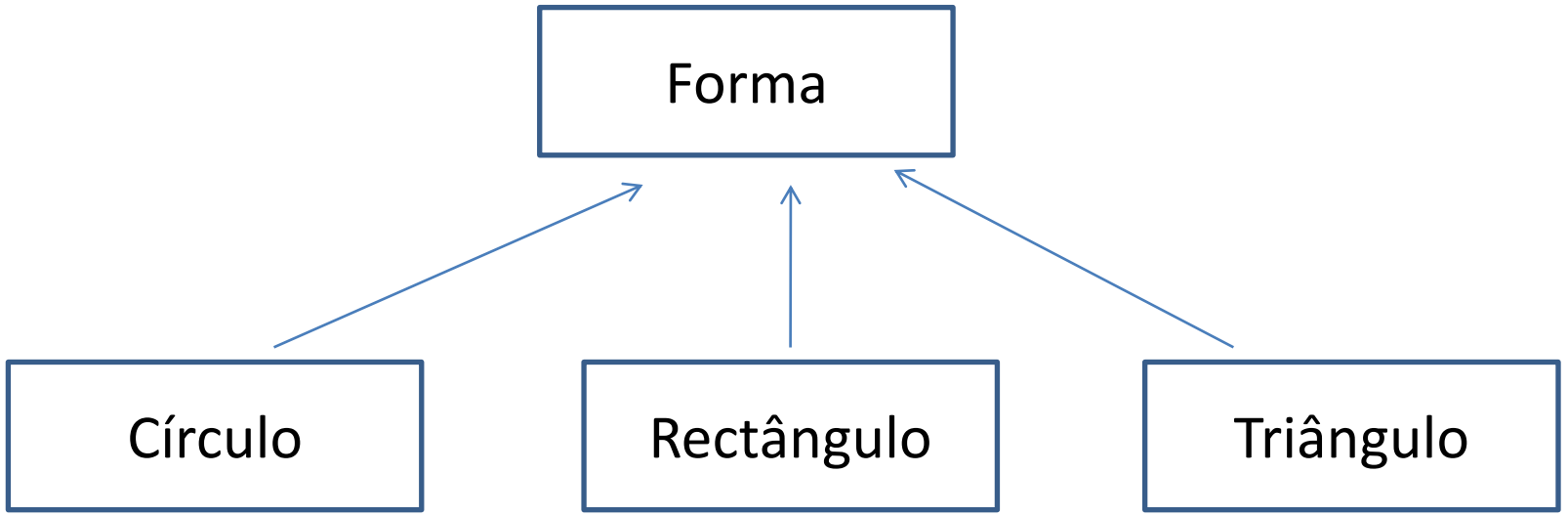
```
public abstract class Forma{
        public abstract double area();
        public abstract double perimetro();
}
```

- Não é possível criar instâncias de uma classe abstracta;
- Mecanismo de herança mantém-se;
- Princípio da substitutividade mantém-se;
- Se uma subclasse de uma classe abstracta implementar todos os métodos, passará a ser uma classe concreta (não abstracta).

Para que servem?

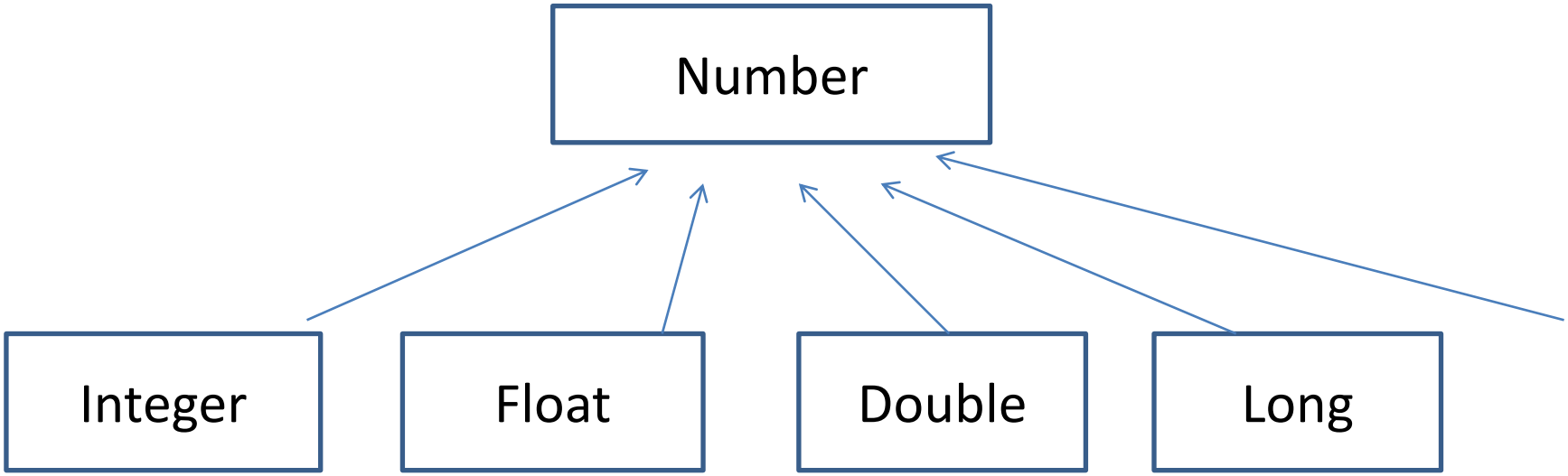
Definir uma linguagem comum a um conjunto de classes que herdam a classe abstracta.

Exemplos:




```
public  class Circulo extends Forma {
 ...
}
...
```

Exemplos:



Notas:

- *Variáveis não são abstractas;*
- *Construtores não são abstractos;*
- *Métodos de classe não são abstractos;*
- *Métodos privados não são abstractos.*

Interfaces (em Java)

"Interface":

- especificação sintáctica de um conjunto de métodos e constantes

Permite definir um comportamento comum a duas ou mais classes que não possuam qualquer relação hierárquica entre si

Declaração de uma interface (exemplo):

```
public abstract interface Ordem{
        public abstract boolean igual (Ordem elemento);
        public abstract boolean maior (Ordem elemento);
        public abstract boolean menor (Ordem elemento);
}
```

Uma interface é (implícita e) obrigatoriamente abstracta.

Os métodos declarados numa interface são (implícita e) obrigatoriamente públicos e abstractos.

Uma classe que implemente uma dada interface tem obrigatoriamente que implementar todos os métodos declarados na interface.

```
public class MyInteger implements Ordem{
        ...
                public boolean igual (Ordem e){…}
                public boolean maior (Ordem e){…}
                public boolean menor (Ordem e){…}
// outros métodos
}
```

Todas as classes que implementam a interface Ordem têm em comum o comportamento definido em Ordem.

As interfaces têm a sua própria hierarquia:

```
public interface Amovivel {
        void movimento ( double x, double y);
}

public interface ComMotor extends Amovivel

        public static final int limiteVel = 120;
        public abstract String motor();
}
```

As constantes declaradas numa interface são implícita e obrigatoriamente: *public static final*

-Uma classe que implemente a interface ComMotor terá obrigatoriamente que implementar:

- todos os métodos da interface e
- todos os métodos de todas as superinterfaces

```
public class Veiculo implements ComMotor {
  …
  public String motor () {…}

  public void movimento (double x, double y){ …}

  …
}
```

Uma interface pode ser sub-interface de várias interfaces:

```
public interface Transformável extends
                                    Escalável,  Rodavel, Desenhavel {
...

}
```

A interface Transformavel herda todas as definições sintácticas das 3 interfaces especificadas.

➔ Mecanismo de herança múltipla

Classes Abstractas versus Interfaces

- uma classe abstracta pode ter métodos implementados

- *numa interface todos os métodos são abstractos*

- uma subclasse de uma classe abstracta pode ser ou não uma classe abstracta

- *numa subinterface todos os métodos são abstractos*

Classes Abstractas versus Interfaces

- uma classe abstracta pode ser usada para escrever software genérico, cada subclasse vai fazendo a sua implementação num processo de especialização sucessiva.

- uma interface serve para especificar um comportamento comum a todas as classes que a implementam.

Compatibilidade entre Interfaces e Classes

Seja a interface Ordem:

```
public interface Ordem{
                boolean igual (Ordem  elemento);
                boolean maior (Ordem  elemento);
                boolean menor (Ordem  elemento);
        }
```

Compatibilidade entre Interfaces e Classes

Seja a interface Ordem:

```
public interface Ordem{
                boolean igual (Ordem  elemento);
                boolean maior (Ordem  elemento);
                boolean menor (Ordem  elemento);
        }
```

Identificadores de interfaces podem ser usados na declaração de variáveis:

```
                Ordem ord1, ord2;
```

- os referências ord1 e ord2 podem ser associadas a quaisquer instâncias que implementem a interface Ordem.

Supondo a classe ClasseA:

```
public class ClasseA implements Ordem {

// implementação dos métodos da interface Ordem (igual, maior e menor)


// outros métodos

public  void m1 () {   …  }
public  void m2 () {   …  }
…

}
```

e a declaração:

```
ClasseA  A1 = new ClasseA();


ord1 = A1;                  // correcto

ord1.m1 ();                 //errado – m1() não é método da interface

ord1.m2 ();                 //errado – m2() não é método da interface

ord1. igual (ord2)          // correcto
```

Herança de interfaces

*Exemplos retirados de **[Martins]** "Programação Orientada aos Objectos em Java 2", F. Mário Martins, FCA, Setembro de 2000.*

O que acontece quando há sobreposição de constantes e métodos?

1) Herança simples de constantes

```
public interface  I1 {

  // constantes
        int x = 1;
        int y = 2;
        int z = 3;

// métodos
        int soma();

}
```

```
public interface I2 extends I1 {

        // constantes
        int x = 10;
        int y = 20;


}
//   as constantes podem ser
 //  redefinidas
```

```
public class Classe1
        implements I1 {


  public int soma(){
     return ( x + y + z );
  }


}
```

```
public class Classe2
        implements I2 {


   public int soma(){
        return ( x + y + z );

      }


}
```

```
public class Teste {

        public static void main (String args [] ){

                Classe1 C1 = new Classe1();
                Classe2 C2 = new Classe2();

                System.out.println ( C1.soma() );
                System.out.println ( C2.soma() );
```

Qual o output deste programa?

Resposta:

6

33

Como aceder ao x e ao y da super-interface?

```
public class Classe3 implements I2 {

        public int soma(){
            return ( I1.x + y + z );
        }
}
```

- prefixar a constante com o nome da interface

2) Herança múltipla de constantes

```
public interface I3 extends I1 , I2  {
        // redefine as constantes
        int x = 100;
        int y = 200;
        int z = 300;
                                        // válido
        // métodos
                int soma();       // sobrepõe o método
}

public interface I4 extends I1, I2 {
}                                       //válido
```

```
public class Classe3 implements I3 {
        public int soma() {
                return (x + y + z ) ;       //válido
        }
}


public class Classe4 implements I4 {
        public int soma() {
                return (x + y + z ) ;       // errado - ambíguo
        }
}
```

```
public class Classe4 implements I4 {

        public int soma() {

                return ( I1.x + I2.y + z ) ;          // correcto

        }

}
```

3) Herança simples de métodos

```
public interface  I1 {

        // constantes
        int a = 1;
        int b = 2;

        // métodos
        int soma();
        int prod();
}
```

```
public interface I2 extends I1 {

// constantes
   int b = 20;  //redefinição


   // métodos
   int soma() //sobreposição *
   int soma (int  x);
   int prod (int x);
}
```

* *tal como na herança de classes, a sobreposição de métodos obriga a igual tipo de resultado*

```
public class C1 implements I1 {
        public int soma() { return a + b;}
        public int prod() { return a * b;}
}

public class C2 implements I2 {
        // métodos de I2
        public int soma() { return a + b;}
        public int soma (int x) { return x + a + b; }
        public int prod  (int x) { return a * x * b;}

        //métodos de I1
        public int prod() {return a * b; }
}
```

Qual o output do seguinte programa?

```
public class TesteInterface{
        public static void main (String args [] ) {
                C1 c1 = new C1();
                C2 c2 = new C2();
                System.out.println ( c1.soma() );
                System.out.println ( c1.prod() );
                System.out.println ( c2.soma() );
                System.out.println ( c2.soma(10) );
                System.out.println ( c2.prod() );
                System.out.println ( c2.prod(5) );
        }
}
```

4) Herança múltipla de métodos

```
public interface  I11 {
    // constantes
    int a = 1;
    int b = 2;

    // métodos
    int soma();

    int prod (); **

   float div (int x, int z);
}
```

```
public interface  I22 {
    // constantes
    int b = 20;

    //métodos
    int soma();
    int soma (int x);

    float prod ();  **

    boolean prod (int x, int y);
}
```

```
public interface I33 extends I11, I22 {

        int soma (int x);  //sobreposição

        float prod (int x , int y);
        // errado, sobrepõe o método com um diferente tipo de resultado

        boolean prod (int k);  // sobrecarga


}
```

*** ao implementarmos I33 ocorrerá um erro de compilação: sobreposição com diferentes tipos de resultados.*

Observações:

1) Se uma interface declara um método, esse método sobrepõe todos os métodos com a mesma assinatura nas suas super-interfaces;
Se um método sobrepõe outro, terá que ter o mesmo tipo de resultado que o primeiro.

2) Se um método sobrepõe outro então a sua cláusula *throws* não pode entrar em conflito com a dos métodos que sobrepõe.

3) Os métodos são sobrepostos com "base" na sua assinatura;

Se uma interface declara dois métodos com o mesmo nome, e uma interface sobrepõe um deles, a interface herda na mesma o outro método.

4) Uma interface herda das suas super-interfaces todos os métodos que não sejam sobrepostos por declaração na interface.

5) É possível uma interface herdar mais do que um método com a mesma assinatura, desde que com o mesmo tipo de resultado.

!! Na classe que implementa essa interface ocorrerá um erro.

5) É possível uma interface herdar mais do que um método com a mesma assinatura, desde que com o mesmo tipo de resultado.

!! Na classe que implementa essa interface ocorrerá um erro.

Supondo a classe Aluno, construída nas aulas anteriores,

– Construa uma pequena aplicação que permita as seguintes opções:

1 - Criar aluno
2 – Consultar aluno, dado o seu número
3 – Consultar aluno, dado o seu nome
4 – Listar todos os alunos
5 – Apagar um aluno
6 – Corrigir o nome de um aluno
7 – Guardar alunos num ficheiro

- Os alunos criados deverão ser armazenados num objecto do tipo java.util.Vector<Aluno>.
- A opção 7 deverá escrever num ficheiro (fichAlunos) o vector com os alunos existentes no vector de alunos.
- Quando o programa é executado deverá começar por ler o conteúdo desse ficheiro.

Duas versões:

- *um só programa e uma função para cada opção.*
- *as várias funções numa classe separada.*