

Polimorfismo e Classes Abstratas

Prof. Ms. Peter Jandl Junior

J12B

Linguagem de Programação Orientada a Objetos

Ciência da Computação - UNIP – Jundiaí

polimorfismo

Do grego *poli morfos*, ou seja, muitas formas.

Característica da orientação a objetos que admite múltiplas formas de suas construções.

Na orientação a objetos o polimorfismo se manifesta na sobrecarga, na herança (como mecanismo de especialização), na sobreposição e também no tratamento polimórfico (que provê generalização).

POO::sobrecarga, herança, sobreposição e polimorfismo

- Na POO - Programação Orientada a Objetos são essenciais:
 - A sobrecarga,
 - A herança
 - A sobreposição e
 - O tratamento polimórfico.
- A rigor, todos estes mecanismos são manifestações diferentes do polimorfismo, que, de fato, é a característica chave na Orientação a Objetos.

Sobrecarga

- Mecanismo que possibilita existir métodos diferentes com mesmo nome.

Herança e Sobreposição

- Mecanismo que possibilita compartilhamento dinâmico de código.

Polimorfismo

- Característica que permite coexistência de muitas formas de representação de classes, seus elementos e seus objetos.

Classes abstratas

- Permite maior controle na criação de hierarquias de classes extensíveis.

Sobrecarga

Quando a ideia prevalece sobre a forma!

Sobrecarga de métodos

- Propriedade da orientação a objetos que permite a existência de dois ou mais métodos com o mesmo nome em uma classe, desde que possuam assinaturas diferentes.
- A sobrecarga de métodos é uma das manifestações do polimorfismo dentro da orientação a objetos.
- Também é conhecida como *method overload*.

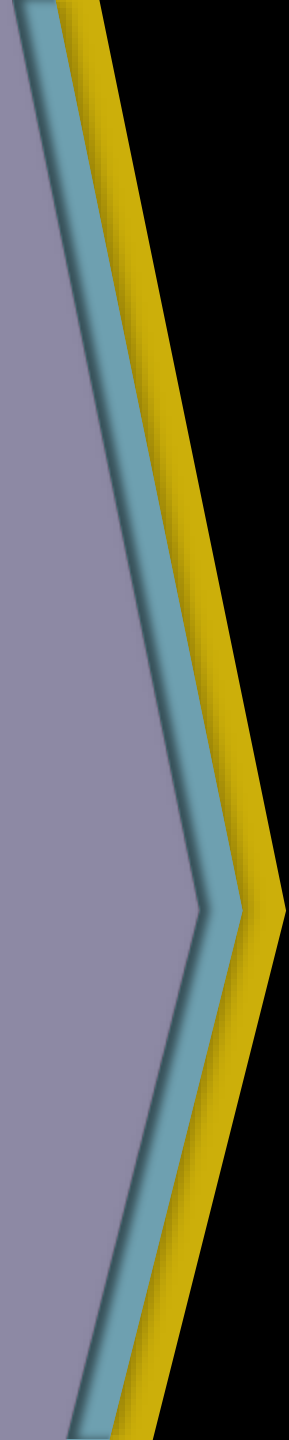
Não confundir com
method override.

Sobrecarga de métodos

- A *assinatura diferente* é o que possibilita esta diferenciação.
- A existência de métodos com mesmo nome e assinatura diferentes significa maneiras distintas de realizar uma mesma operação.
- Pode ser empregada para criação de múltiplos construtores na classe (i.e, formas diferentes de obter-se instâncias desta classe).
- Isto proporciona *flexibilidade* para o uso da classe!

Herança e Sobreposição

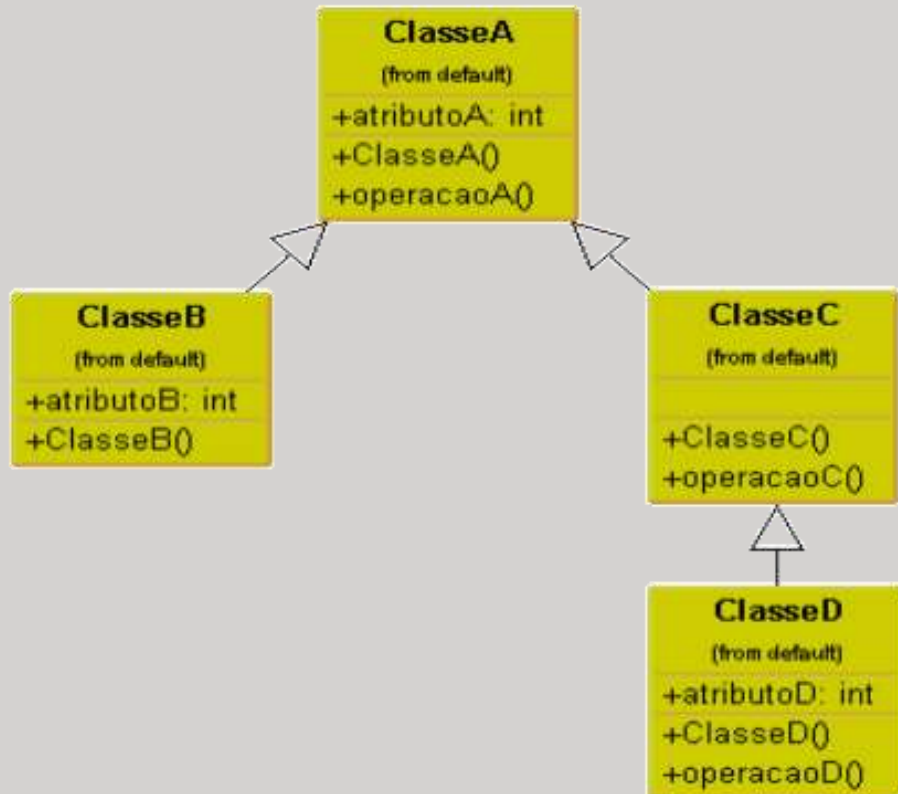
O polimorfismo no caminho da especialização!



Herança

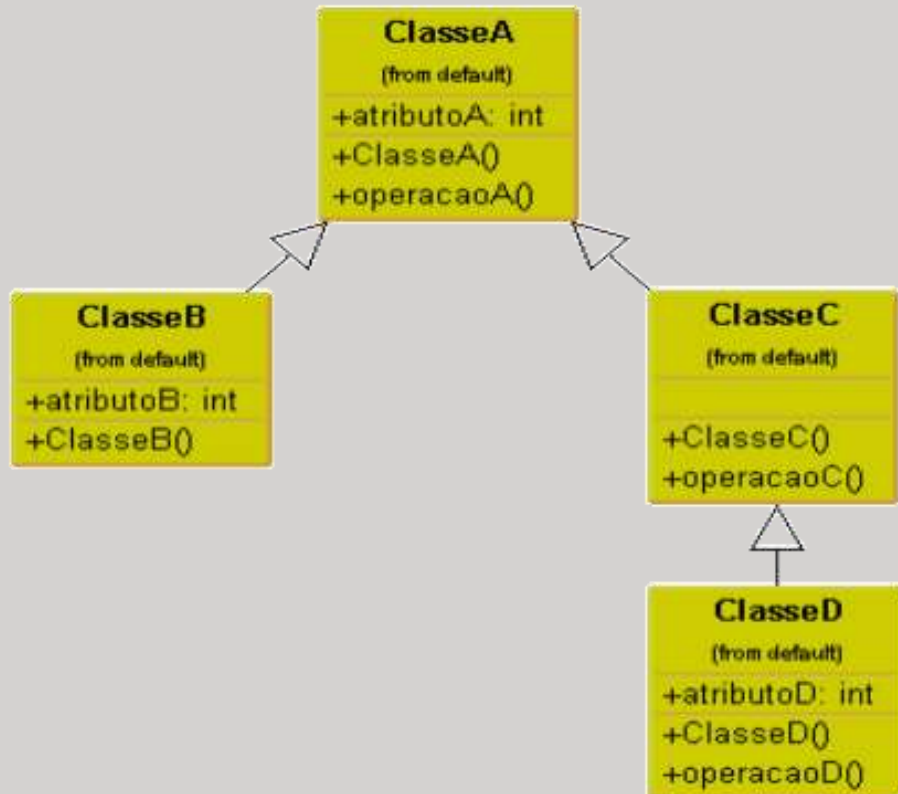
- ❑ Segunda característica mais importante da Orientação a Objetos
- ❑ Significa a construção de novos tipos de dados (classes) baseada em outros tipos já existentes, onde:
 - ❑ as características ancestrais são compartilhadas pelos descendentes e
 - ❑ novas características são adicionadas nestes.
- ❑ Possibilita a *especialização* das classes.

Herança & Hierarquias de classes



- Classe existente (*superclasse*, *classe base* ou *classe-pai*) dá origem a nova classe (*subclasse*, *classe derivada* ou *classe-filha*).
- Isto permite criar *hierarquias* (ou famílias) de classes.
- Possibilita *compartilhar* a implementação, pois atributos e operações *públicas* e *protegidas* são acessíveis para os descendentes da classe.

Herança & Hierarquias de classes



- Atributos e operações **privadas** são *exclusivos* de cada classe.
- Os **construtores** *nunca são compartilhados*.
- Herança simplifica projeto e permite reuso do código.
- A existência de relações de herança é expressa pela resposta afirmativa a questão "*é um?* | *é do tipo?*";

Herança::acessibilidade de membros

Especificador	Acessibilidade de membros			
	Implementação Superclasse	Instâncias Superclasse	Implementação SubClasse	Instâncias SubClasse
private	sim	não	não	não
protected	sim	não	sim	não
public	sim	sim	sim	sim

Membros protegidos constituem uma espécie de herança de programador, isto é, só está disponível para *implementações* de subclasses.

Herança::aplicações

Extensão

- A Herança permite a criação de novas subclasses que ampliam as operações e atributos existentes na superclasse.

Restrição

- A Herança também permite que as funcionalidades da superclasse sejam alteradas em subclasses.

A **Herança** é um mecanismo de *especialização*, ou seja, permite a construção de novas classes derivadas que possuem características especiais em relação a classe base.

Herança:sobreposição de métodos

- A sobreposição (ou substituição) de métodos consiste na implementação de método na subclasse com a mesma assinatura de outro existente na superclasse.
- Permite dotar a subclasse com implementação distinta da superclasse, mas mantendo sua interface (o que facilita seu uso por meio do polimorfismo).
- Também é conhecida como *method override*.

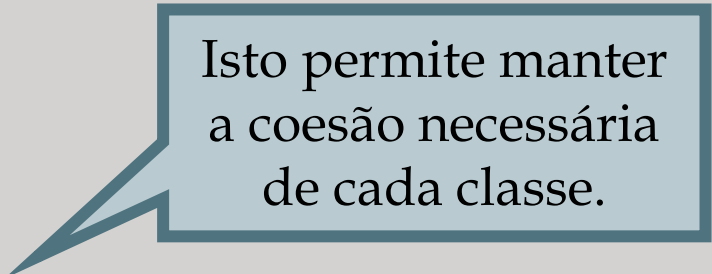
Method overload é a sobrecarga de métodos *intraclasse*.

Não confundir com *method overload*.

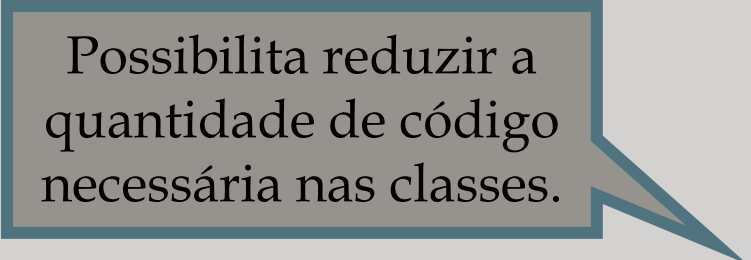
Method override é a sobreposição de métodos *interclasses*.

Herança::projeto de classes

- O mecanismo da herança pode ser melhor aproveitado quando:
 - A superclasse contém todos os elementos (atributos e métodos) comuns de um conjunto de classes.
 - Cada subclasse contém apenas os elementos específicos que a caracterizam.



Isto permite manter a coesão necessária de cada classe.



Possibilita reduzir a quantidade de código necessária nas classes.

Herança::projeto de classes

- O mecanismo da herança pode ser melhor aproveitado quando:
 - Características comuns e específicas podem ser identificadas com uso de uma tabela “classes versus atributos,métodos” para facilitar a fatoração.

Classes são usualmente os **substantivos** principais da narrativa descritiva do problema!

Atributos são geralmente os **substantivos** secundários da narrativa descritiva do problema,enquanto métodos são os **verbos**!

CLASSES ABSTRATAS

Para quando sabemos o que queremos,
mas não sabemos como fazer!

Classes Abstratas

Garantindo uma *interface comum* entre tipos diferentes.

- Existem casos onde:
 - não se deseja ou
 - não é possível implementar métodos em uma classe,
 - mas necessário indicar sua obrigatoriedade.
- Em outras situações são observadas operações diferentes com a mesma denominação (p.e., salário calculados de maneiras diferentes para tipos distintos de funcionários).

Classes Abstratas

- Ainda temos situações onde não se deseja a existência de objetos de um determinado tipo, que, no entanto, constitui a superclasse de outras classes cujos objetos são necessários.

Classes Abstratas

- Em todos estes casos, observa-se a necessidade da existência de uma semântica comum à estas classes para:
 - indicar as operações necessárias,
 - e possibilitar o tratamento polimórfico de seus objetos.
- As classes abstratas endereçam exatamente estes tipos de problema.

Classes & Métodos Abstratos

- Métodos abstratos são aqueles cuja implementação será *adiada* para subclasses. São declarados como protótipos e recebem o modificador *abstract*:
public abstract double area();
- Classes que contêm métodos abstratos são consideradas *abstratas*.
- Classes que não contêm métodos abstratos podem ser declaradas abstratas por meio do modificador *abstract*:
public abstract class Forma { ... }

Classes Abstratas & Classes Concretas

- Classes abstratas são aquelas que *não podem* ter objetos instanciados.
 - Classes concretas são aquelas que *podem* ter objetos instanciados.
-
- Uma subclasse de classe abstrata se torna concreta quando supre os métodos abstratos necessários, podem assim ter objetos instanciados.

Classes Abstratas::implicações

1. **Não é possível instanciar objetos de classes abstratas**, pois existem métodos incompletos (não implementados, definidos como **abstract**) ou a própria classe está marcada como **abstract**.
2. **Obriga que as subclasses implementem construtores** para suportar os construtores parametrizados da superclasse abstrata.

Classes Abstratas::implicações

3. **Obriga que suas subclasses implementem métodos abstratos existentes** para que tornem classes concretas, criando uma espécie de contrato futuro.
4. **Define uma semântica mínima comum** a todos os participantes de uma hierarquia de classes, provendo tratamento polimórfico.

Classes Abstratas

- Constituem alternativa atraente no projeto de hierarquia de classes, pois:
 - Permitem a criação de tipos comuns que simplificam a implementação de classes;
 - Possibilitam a adição de novos tipos a um projeto existente, facilitando os procedimentos de alteração devido à necessidade de correções ou demanda por evolução do sistema.

Classes Abstratas::Aplicação

O uso essencial das classes abstratas no projeto de hierarquia de classes.

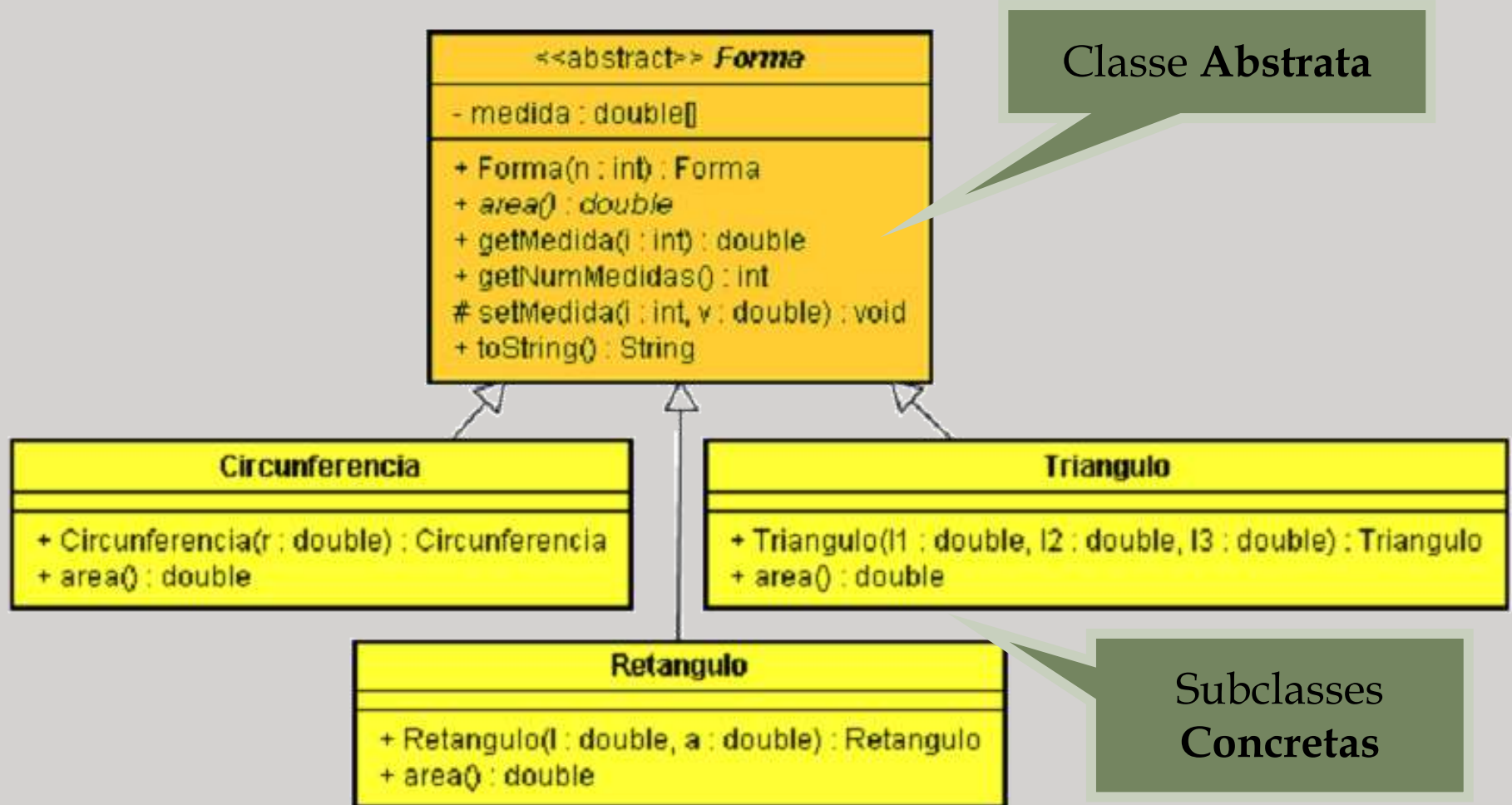
Classes Abstratas::exemplo

- Uma classe **Forma** poderia reunir os elementos comuns a formas geométricas planas simples sem, no entanto, ter uma implementação completa. Ou seja, inclui:
 - Armazenamento das medidas dos lados da forma;
 - Operações genéricas *get* e *set* para administrar as medidas dos lados; e
 - Operações específicas como *area()* ou *perimetro()*.

Classes Abstratas::exemplo

- Novas classes concretas, i.e., formas geométricas específicas podem ser criadas como subclasses de **Forma**, tal como:
 - Circunferencia
 - Retangulo
 - Quadrado
 - Triangulo

Hierarquia de classes abstratas & concretas



Classes Abstratas::exemplo

Forma.java

```
public abstract class Forma {  
    private double medida[];  
  
    public Forma(int lados) {  
        medida = new double[lados];  
    }  
    public int getNumLados( ) {  
        return medida.length;  
    }  
}
```

```
public double getMedida(int i) {  
    if (i<0 || i>medida.length) {  
        throw new RuntimeException();  
    }  
    return medida[i];  
}  
public void setMedida(int i, double m)  
{  
    if (i<0 || i>medida.length) {  
        throw new RuntimeException();  
    } else if (m<0) {  
        throw new RuntimeException();  
    }  
    medida[i] = m;  
}  
// método(s) abstrato(s)  
public abstract double area ( );  
}
```

Classes Abstratas: exemplo

Circunferencia.java

```
public class Circunferencia  
    extends Forma {
```

```
    public Circunferencia (  
        double raio) {  
        super(1);  
        setMedida(0, raio);  
    }
```

Construtor obrigatório para suprir número de lados requisitado pelo construtor da superclasse **Forma**.

Método que sobrepõe definição abstrata existente na superclasse **Forma**.

```
        public double area ( ) {  
            return Math.PI *  
                Math.pow(getMedida(0), 2);  
        }  
    }
```

Novas classes requerem pouco esforço de codificação, pois compartilham implementação de **Forma**.

Classes Abstratas: exemplo

Retangulo.java

```
public class Retangulo
    extends Forma {
    public Retangulo (
        double alt, double larg) {
        super(2);
        setMedida(0, alt);
        setMedida(1, larg);
    }
```

Construtor obrigatório para suprir número de lados requisitado pelo construtor da superclasse **Forma**.

Método que sobrepõe definição abstrata existente na superclasse **Forma**.

```
    public double area ( ) {
        return getMedida(0) *
            getMedida(1);
    }
```

Novas classes requerem pouco esforço de codificação, pois compartilham implementação de **Forma**.

Classes Abstratas:exemplo

Triangulo.java

```
public class Triangulo
    extends Forma {
    public Triangulo (double l1,
        double l2, double l3) {
        super(3);
        setMedida(0, l1);
        setMedida(1, l2);
        setMedida(2, l3);
    }
```

Construtor obrigatório para suprir número de lados requisitado pelo construtor da superclasse **Forma**.

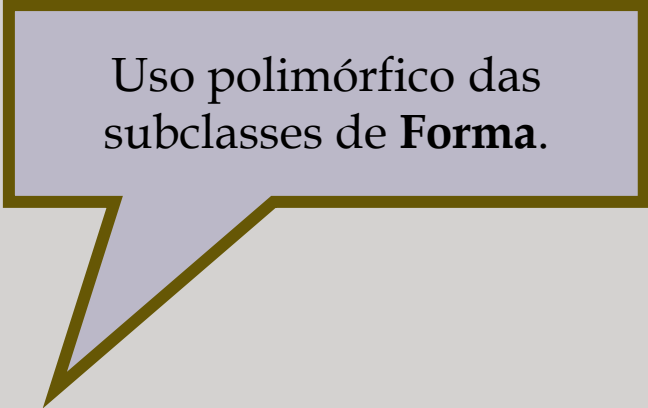
Método que sobrepõe definição abstrata existente na superclasse **Forma**.

```
    public double area ( ) {
        return ... ;
    }
```

Novas classes requerem pouco esforço de codificação, pois compartilham implementação de **Forma**.

Classes Abstratas

```
public class TestaFormas {  
    public static void main (String a[]) {  
        Circunferencia c = new Circunferencia (1.5);  
        System.out.println("areaCirc = "+ c.area());  
        Retangulo r = new Retangulo (2.5, 4.0);  
        System.out.println("areaRet = "+ r.area());  
        Forma formas[] = new Forma [3];  
        formas[0] = c;  
        formas[1] = r;  
        formas[2] = new Triangulo (1.2, 1.5, 1.8);  
        for (int i=0; i<formas.length; i++) {  
            System.out.println(formas[i]);  
            System.out.println("area = "+ formas[i].area());  
        }  
    }  
}
```



Uso polimórfico das subclasses de **Forma**.

Classes Abstratas

- Uma nova classe pode ser adicionada à hierarquia de duas maneiras:
 - Tomando uma classe concreta existente como base; ou
 - Tomando a raiz abstrata como base.

- Nos dois casos:
 - Nenhuma das classes existentes da hierarquia exigirão alterações.
 - Apenas classes "interessadas" nos novos tipos deverão ser modificadas, possivelmente com impacto mínimo ou bastante específico.

Classes Abstratas: exemplo

Quadrado.java

```
public class Quadrado  
    extends Retangulo {  
    public Quadrado  
        (double lado) {  
        super(lado, lado);  
    }  
}
```

Construtor obrigatório para suprir número de lados requisitado pelo construtor da superclasse **Retangulo**

Método **area()** não precisa sobreposição, pois está correto na superclasse **Retangulo**.

```
}
```

Novas classes requerem pouco esforço de codificação, pois compartilham implementação de **Forma**.

Classes Abstratas

```
public class TestaFormas {  
    public static void main (String a[]) {  
        Circunferencia c = new Circunferencia (1.5);  
        System.out.println("areaCirc = "+ c.area());  
        Retangulo r = new Retangulo (2.5, 4.0);  
        System.out.println("areaRet = "+ r.area());  
        Forma formas[] = new Forma [4];  
        formas[0] = c;  
        formas[1] = r;  
        formas[2] = new Triangulo (1.2, 1.5, 1.8);  
        formas[3] = new Quadrado (2.3);  
        for (int i=0; i<formas.length; i++) {  
            System.out.println(formas[i]);  
            System.out.println("area = "+ formas[i].area());  
        }  
    }  
}
```

Uso de nova subclasses de **Forma** traz impacto mínimo ao programa.

Uso polimórfico das subclasses de **Forma**.

Uso de classes abstratas na API Java

- A API Java também faz uso das classes abstratas, p.e, no Swing e no *framework* de Coleções:
 - Swing
 - javax.swing.AbstractButton
 - javax.swing.table.AbstractTableModel
 - Coleções:
 - java.util.AbstractCollection,
 - java.util.AbstractSet,
 - java.util.AbstractList
 - java.util.AbstractMap

Recomendações de Estudo



- Java – Guia do Programador, 3ª Edição, P. JANDL Jr, Novatec, 2015.
- Java 6- Guia de Consulta Rápida, P. JANDL Jr, Novatec, 2008.
- Java 5- Guia de Consulta Rápida, P. JANDL Jr, Novatec, 2006.
- Introdução ao Java, P. JANDL Jr, Berkeley, 2002.