UNIVERSIDADE DO ESTADO DE SANTA CATARINA
DCC – Departamento de Ciência da Computação
Joinville – SC

CURSO: Bacharelado em Ciência da Computação
DISCIPLINA: POO0001 – Programação Orientada a Objetos
PROFESSOR: Diego Buchinger
AULA 06 – Classes Abstratas e Interface

1. CLASSES ABSTRATAS

O uso do conceito de herança, visto na última aula, pode ser muito útil para reaproveitamento de código mas, as vezes, ao criar uma superclasse para se reaproveitar código não queremos que esta possa ser instanciada. Por exemplo, se em um sistema criamos uma superclasse Cliente para agrupar atributos e métodos em comum entre clientes pessoa física e pessoa jurídica, pode ser indesejável que seja possível instanciar um objeto Cliente, que não seja nem pessoa física nem pessoa jurídica. É exatamente para estes tipos de situações que existem as classes abstratas!

Uma classe abstrata pode incluir atributos e métodos como uma classe normal, contudo ela pode incluir também métodos abstratos e, principalmente, não pode ser instanciada. Os métodos abstratos mencionados são assinaturas de métodos que não podem ser implementadas nas classes abstratas, mas devem ser obrigatoriamente implementados nas classes derivadas (subclasses) sobrescrevendo-os com uma implementação. No Java, a palavra-chave *abstract* é utilizada para representar métodos e classes abstratas, devendo ser usada após o indicador de encapsulamento e antes do tipo de retorno.

Para exemplificar o uso de classes abstratas considere a seguinte tentativa de representar diferentes tipos de animais: cachorro, vaca e tucano. Precisa-se saber o nome e a idade destes animais mas, além disso, para os cachorros é interessante saber ainda o seu gênero, para as vacas precisa-se saber o seu peso e o número total de litros de leite que elas já produziram no total, e para os tucanos é preciso saber a cor de sua plumagem. Para aproveitar os dados e métodos que estes animais possuem em comum elaborou-se uma classe abstrata genérica, e cada animal foi especializado em uma classe derivada. Uma implementação é proposta no próximo quadro.

Na Figura 1 mostra-se o aviso de erro/alerta que a IDE Netbeans fornece quando definimos que a classe Dog herda da classe Animal, sendo que esta superclasse já está implementada e possui a definição de um método abstrato *toString()*. A mensagem mostra que a classe Dog deve ser abstrata ou deve sobrescrever o método *toString()*.

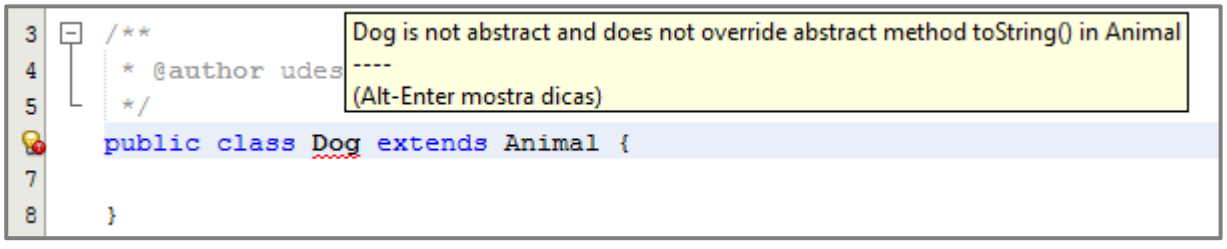

FIGURA 1 – Mensagem de erro/alerta quando há um método abstrato que não foi sobrescrito.

Arquivo: Animal.java

```java
package abstrata;

public abstract class Animal {
    public String name;
    private int age;

    public Animal(String name){
        this.name = name;
        age = 0;
    }
    public void anniversary(){ age += 1; }
    public int getAge(){ return age; }

    public abstract String toString();
}
```

Arquivo: Dog.java

```java
package abstrata;

enum Gender { Male, Female };

public class Dog extends Animal {
    private Gender gender;

    public Dog(String name, Gender gender){
        super(name);
        this.gender = gender;
    }

    @Override
    public String toString() {
        return "Este cachorro se chama: " + name;
    }
    public Gender getGender(){
        return gender;
    }
}
```

Arquivo: Cow.java

```java
package abstrata;

public class Cow extends Animal{
    static float totalMilkProduced = 0;
    int weight;

    public Cow(String name, int weight){
        super(name);
        this.weight = weight;
    }

    @Override
    public String toString() {
        return "Isto é uma vaca e seu nome é: " + name +
                "\nEla possui " + weight + " kg";
    }
```

```
17.
18.       public void produceMilk(float producedMilk){
19.           Cow.totalMilkProduced += producedMilk;
20.       }
21.   }
```

Arquivo: Toucan.java

```
 1.  package abstrata;
 2.
 3.  public class Toucan extends Animal {
 4.       String plumageColor;
 5.
 6.       public Toucan(String name, String plumage){
 7.           super(name);
 8.           plumageColor = plumage;
 9.       }
10.
11.       @Override
12.       public String toString() {
13.           return "Isto é uma Tucano chamado de " + name +
14.                   "\nque possui plumagem " + plumageColor;
15.       }
16.   }
```

2. INTERFACE

Outro recurso muito útil proveniente da orientação a objetos é o conceito de interface – CUIDADO, não confundir com interface gráfica. Uma interface modela um comportamento esperado, mas não o implementa. Ela pode ser entendida como uma classe abstrata que contêm apenas constantes e métodos abstratos.

Na linguagem de programação Java, as interfaces são identificadas pela palavra *interface* – que substitui a palavra-chave *class* – e podem ser utilizadas inclusive para viabilizar herança múltipla. Na interface só podem ser incluídas constantes estáticas e públicas, e métodos abstratos públicos; por esse motivo, não é necessário utilizar os especificadores *public*, *static* e *final*. Lembre-se que constantes devem ter um valor atribuído na sua declaração.

Conforme dito anteriormente, uma interface modela um comportamento esperado. Este comportamento é efetivamente especificado e realizado pelas classes que "implementam" a interface, ou seja, que implementam os métodos que foram definidos por ela. Para indicar que uma classe implementa uma interface em Java, utilizamos a palavra-chave *implements*. Como uma classe pode implementar mais de uma interface, devemos separar o nome das interfaces por vírgula, caso isso ocorra. Veja o exemplo mostrado no próximo quadro que define duas interfaces e uma classe que as implementa.

CURIOSIDADE: No quadro apresentado a seguir perceba que o método *toString()* foi marcado como reescrito (@Override) tanto na classe Point2D como em Rectangle, contudo eles não vem de nenhuma herança explícita. Então por que do uso da marcação de reescrita? Isto vem da herança implícita da classe Object, que possui um método *toString()*. Inclusive, ao sobrescrever este método, é possível usar um objeto da classe nos métodos de impressão, os quais invocaram a função *toString()*. Que lindo! =)

Arquivo: Movable.java

```java
package interfaces;

public interface Movable {
    void moveVertical(int units);
    void moveHorizontal(int units);
}
```

Arquivo: Relatable.java

```java
package interfaces;

public interface Relatable {
    /**
     * this (object calling isLargerThan) and other
     * must be instances of the same class
     * @param other the object to compare to
     * @return returns true if this is larger than other.
     */
    boolean isLargerThan(Relatable other);
}
```

Arquivo: Point2D.java

```java
package interfaces;

public class Point2D {
    public int x, y;
    public Point2D(){
        x = y = 0;
    }
    public Point2D(int x, int y){
        this.x = x;
        this.y = y;
    }

    @Override
    public String toString(){
        return "x: " + x + " / y: " + y;
    }
}
```

Arquivo: Rectangle.java

```java
package interfaces;

public class Rectangle implements Movable, Relatable {
    Point2D center;
    int width, height;

    public Rectangle(int w, int h){
        width = w;
        height = h;
        center = new Point2D();
    }

    public int getArea(){
        return width * height;
    }
```

```
16.
17.       @Override
18.       public String toString(){
19.           return "Retângulo Info:\nCentro: " + center +
20.                "\nLargura: " + width + "  |  Altura: " + height;
21.       }
22.
23.       @Override
24.       public void moveVertical(int units) {
25.           center.y += units;
26.       }
27.
28.       @Override
29.       public void moveHorizontal(int units) {
30.           center.x += units;
31.       }
32.
33.       @Override
34.       public boolean isLargerThan(Relatable other) {
35.           if( !(other instanceof Rectangle) )
36.               return false;
37.           Rectangle aux = (Rectangle) other;
38.           if( this.getArea() > aux.getArea() )
39.               return true;
40.           return false;
41.       }
42.   }
```

Arquivo: Main.java

```
 1.  package interfaces;
 2.
 3.  public class Main {
 4.      public static void main(String[] args){
 5.          Rectangle myRectangle = new Rectangle(5,5);
 6.          myRectangle.moveVertical( 5 );
 7.          myRectangle.moveHorizontal( -2 );
 8.          System.out.println( myRectangle );
 9.
10.          Relatable rectangleB = new Rectangle( 4, 6 );
11.          int area = ((Rectangle) rectangleB).getArea();
12.          System.out.println("\nÁrea do novo retângulo: "+area);
13.
14.          if( myRectangle.isLargerThan( rectangleB ) )
15.              System.out.println("Primeiro retângulo é maior!");
16.          else
17.              System.out.println("Segundo retângulo é maior!");
18.      }
19.  }
```

Outro detalhe que merece ser destacado é a possibilidade de uso do conceito de polimorfismo usando interfaces. Assim como uma classe derivada pode receber *upcasting* para sua superclasse, uma classe que implementa uma interface também pode receber *upcasting* para a interface a qual implementa. Isto é, um objeto de uma classe que implementa uma interface, pode se passar pela interface através de *upcasting* se isto for vantajoso. O contrário, *downcasting*, também pode ocorrer, lembrando-se de ter o cuidado de verificar o tipo da instância.

No exemplo apresentado no quadro anterior, a propriedade de *upcasting* é usada em dois momentos. Primeiro na linha 10 da classe Main, quando o segundo retângulo é criado, definindo que é do tipo (interface) relacionável (Relatable), mas é instanciado como Retângulo (até porque não se pode criar uma instância de interface). Segundo, na linha 14 da classe Main, uma chamada do método *isLargerThan(Relatable)*, que é definida na linha 34 da classe Rectangle. Note que ao passar um retângulo, ocorre *upcasting* e ele é recebido como um objeto relacionável (Relatable).

De maneira semelhante, a propriedade de *downcasting* também é usada pelo menos em dois momentos no programa apresentado no quadro anterior. Primeiro na linha 11 da classe Main, de modo a identificar que o objeto é um retângulo e poder utilizar o método *getArea()* definido na classe Rectangle. Segundo, na linha 37 da classe Rectangle. Note que neste caso foi utilizada uma verificação da instância do objeto na linha 35 da mesma classe, evitando que se outro objeto que implementa a interface Relatable seja passado como parâmetro, este não gere um erro/exceção.

2.1 Principais diferenças entre interfaces e classes abstratas

Interfaces e classes abstratas compartilham algumas características, mas possuem suas peculiaridades próprias. A seguir é apresentada uma lista das principais diferenças entre elas:

- As classes abstratas podem conter atributos usuais não estáticos, além de métodos ordinários com implementação. As interfaces, por outro lado, só podem ter constantes e métodos abstratos públicos, nos quais inclusive não é necessário utilizar os identificadores: *final* e *abstract*, respectivamente, sendo que estes são inferidos automaticamente;
- Classes abstratas geralmente são utilizadas para compartilhar e reaproveitar pedaços de código, ao passo que interfaces costumam ter o objetivo de determinar um comportamento mínimo esperado pelas classes que a implementam;
- Apesar de ser possível a criação de uma classe abstrata somente com constantes e métodos abstratos, esta seria mais bem especificada como uma interface.

Exercícios

1) Escreva uma classe abstrata `Shape` com as seguintes propriedades: um atributo `shapeName` do tipo `String`, um método abstrato `double area()` e um método `toString()` que retorna o nome da forma.

Escreva três classes derivadas de `Shape`: `Sphere`, `RectangularPrism`, e `Cylinder`, sendo que a esfera possui um atributo **raio**, o prisma retangular possui **altura**, **largura** e **comprimento**, e o cilindro possui **raio** e **altura**. Implemente os métodos de área para cada um deles, lembrando que a área de superfície da esfera é dada pela fórmula: $4\pi r^2$ (onde r é o raio), a área do prisma retangular é dada por: $2(ab) + 2(ac) + 2(bc)$ (onde a é a altura, b é a largura e c é o comprimento) e a área do cilindro é dada pela fórmula: $2\pi r^2 + 2\pi rh$ (onde r é o raio e h é a altura). Implemente ainda o método `toString()` para as três formas, usando o método `toString()` da superclasse [use: super.toString()] e adicionando também os valores dos atributos. Veja abaixo um exemplo do construtor da esfera:

```
public Sphere(double r){
    super("Sphere");
    radius = r;
}
```

A seguir (próxima página) é apresentada a classe `Paint` que representa um tipo de tinta. Corrija o método *amount()* de tal forma que a quantidade correta de tinta necessária seja retornada (OBS: utilize o atributo *coverage* que indica quantos cm² podem ser pintados com um galão/latão de tinta e, obviamente a área da forma que foi passada como parâmetro).

Na próxima página também é apresentado a classe `PaintThings` que contem o programa principal que seria responsável por gerar as formas e computar a quantidade de tinta necessária. O programa já instancia um tipo de tinta; cabe a você completar o programa de modo a:

(1) Instanciar três formas: um retângulo prismático de 20 x 35 x 10, uma grande esfera (bigBall) com um raio de 15, e um tanque cilíndrico de raio 10 e altura 30;

(2) Utilize os métodos apropriados para atribuir os valores de tintas necessários as variáveis corretas.

```java
//*******************************************************
// Paint.java
// Representa o tipo de tinta que tem uma área de cobertura fixa
// por galão/latão. As medidas são em cm².
//*******************************************************
public class Paint{
   private double coverage; //número de cm² por galão/latão

   /**
    * Construtor: Monta o objeto de pintura.
    */
   public Paint(double c){
        coverage = c;
   }

   /**
    * Retorna a quantidade de tinta (número de galões)
    * necessário para pintar a forma passada como parâmetro
    */
   public double amount(Shape s){
        System.out.println ("Computing amount for " + s);
        return 0;
   }
}

//*************************************************************
// PaintThings.java
// Computa a quantidade de tinta necessária para pintar
// várias coisas
//*************************************************************
import java.text.DecimalFormat;
public class PaintThings{
    public static void main (String[] args){
        final double COVERAGE = 350;
        Paint paint = new Paint(COVERAGE);
        double deckAmt, ballAmt, tankAmt;

        // Instancia as três formas a serem pintadas
        // ...
        // Computa a quantidade de tinta necessária para cada forma
        // ...
        // Mostra a quantidade de tina necessária.
        DecimalFormat fmt = new DecimalFormat("0.#");
        System.out.println ("\n# de latões necessários...");
        System.out.println ("Deck: " + fmt.format(deckAmt) );
        System.out.println ("Big Ball: " + fmt.format(ballAmt) );
        System.out.println ("Tank: " + fmt.format(tankAmt) );
    }
}
```

2) Implemente um sistema de agenda que salva contatos seguindo a descrição abaixo:

Crie uma classe denominada `Contato` com pelo menos os seguintes atributos: *identificador*, *nome*, *endereço*, *telefone*, *whatsapp* e *email*. Esta classe deve ter um método construtor que recebe alguns ou todos os atributos (pelo menos os mais importantes). Utilize *getters* e *setters* se necessário para garantir que: o *identificador* seja um número positivo maior que zero, o *nome* não pode ser alterado, o *telefone* tenha entre 9 e 10 dígitos (formato: xxxx-xxxx ou xxxxx-xxxx) e o *email* deve ser vazio ou conter um @ e um ponto (.). Implemente um método toString() que mostra os dados do contato de maneira formatada.

Também crie uma interface chamada `Agenda` com as seguintes assinaturas de métodos:

```
boolean adicionarContato(Contato novoContato);
boolean removerContato(int identificador);
Contato consultarContato(int identificador);
boolean alterarContato(Contato contato);
```

Agora, implemente uma classe `AgendaMemória` que implementa a interface `Agenda`. Essa classe armazenará os contatos usando apenas a memória do computador. Tal classe deve ter um construtor que recebe um valor inteiro simbolizando o tamanho da agenda – crie um vetor do tamanho especificado pelo usuário. Se a adição de um novo contato ultrapassar o limite estabelecido inicialmente, o método deverá retornar apenas um valor falso. Adicionalmente, a classe `AgendaMemória` ainda deve ter um método:
`void limparContatos()` que exclui todos os contatos da lista, e um método:
`Contato consultarNome(String nome)` que consulta e retorna um contato pelo nome passado como argumento.

Implemente também uma outra classe, `AgendaArquivo` que também implementa a interface Agenda. Essa classe armazenará os contatos usando um arquivo. O construtor desta classe deve receber uma String indicando o nome do arquivo a ser criado, ou utilizado para armazenar os dados dos contatos. Adicionalmente, esta classe ainda deve oferecer o seguinte método: `consultarTelefone(String telefone)` o qual consulta e retorna um contato pelo telefone passado como argumento.

NOTA: se o contato procurado nas consulta não for encontrado, deve-se retornar *null*.

Por fim, implemente ainda uma classe Main que faz uso das agendas criadas e de seus métodos. Não será exigido, mas seria interessante elaborar uma interface com o usuário (no console) com um menu, através do qual ele poderá realizar as operações disponíveis. A definição da Agenda a ser usada – se é a AgendaMemória ou AgendaArquivo – pode ser feita antes da compilação, mas o programa deverá prever as opções de funcionalidades adicionais para cada caso.