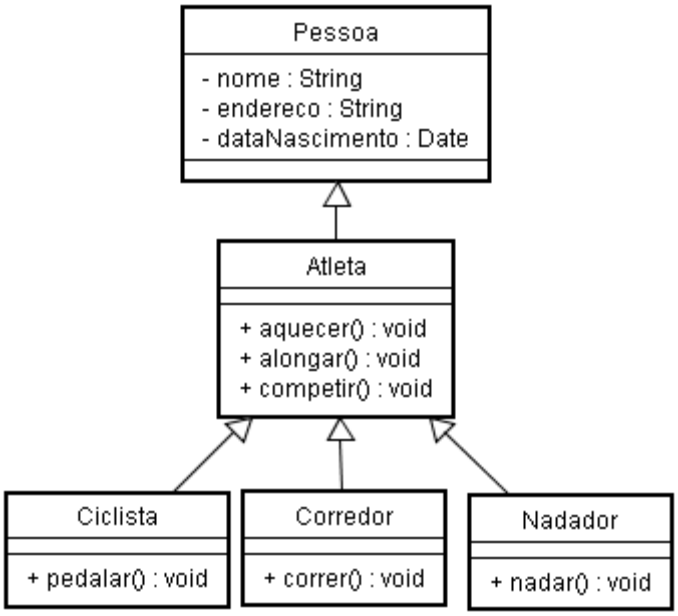
Nome: _____ Data: ____/____/____

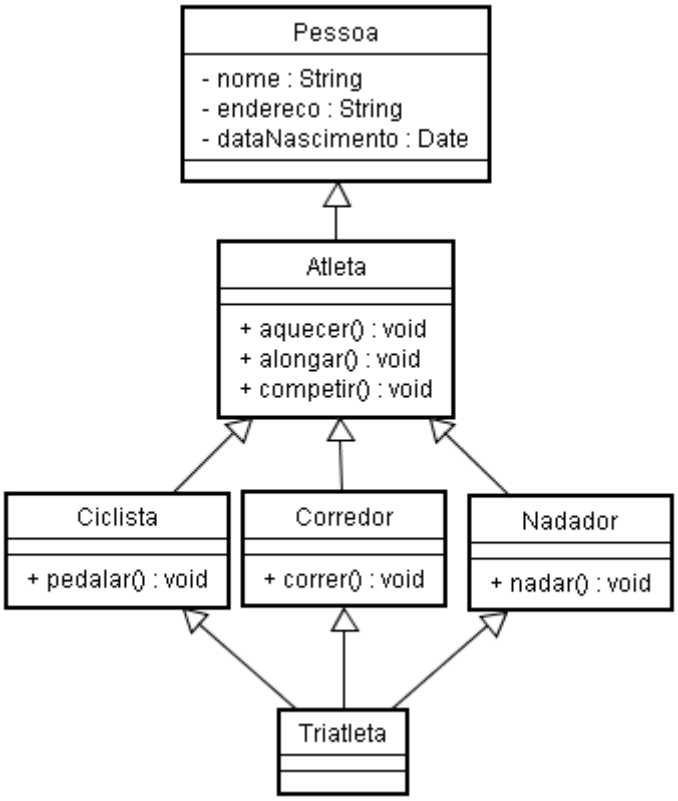
## Interfaces, classes internas e tratamento de eventos

**O problema da herança simples:**

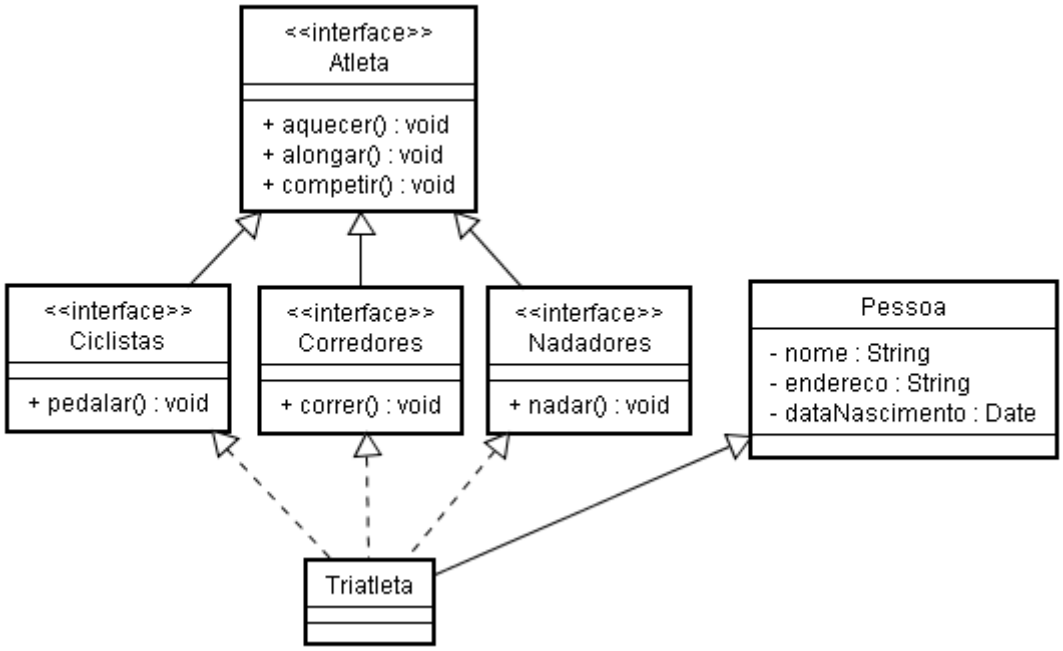
Depois de alguma reflexão mais profunda, ou uma experiência maior na criação de classes, podemos descobrir que a simplicidade da herança simples é restritiva. Particularmente quando o mesmo comportamento deve ser utilizado por diversas classes. Para solucionar esse problema, a linguagem Java possui um mecanismo chamado Interface. Com o uso das interfaces as classes, apesar de herdarem atributos e métodos de somente uma superclasse, podem "herdar" os comportamentos de diversas interfaces. Considere o seguinte exemplo: Um atleta é uma pessoa, ou seja, a classe Atleta herda da classe Pessoa. Um nadador é um atleta, ou seja, a classe Nadador herda da classe Atleta. Da mesma forma, podemos ter as classes Corredor e Ciclista, que também herdam da classe Atleta:




Até aqui, sem grandes problemas, porém imagine a classe Triatleta. O triatleta é um atleta que pedala, corre e nada. Ou seja, ele tem as características das três classes: Ciclista, Corredor e Nadador. Nesse contexto, a nossa classe Triatleta teria que herdar os comportamentos das classes Ciclista, Corredor e Nadador:




É ai que começam os problemas. **Na linguagem Java, a herança é simples, ou seja, o modelo mostrado acima é inválido para implementação utilizando a linguagem Java**. Porém, para suprir esse "falta" e resolver o problema de uma maneira elegante, a linguagem Java conta com as Interfaces:



Com o conceito de Interfaces, a classe Triatleta simplesmente herda da classe Pessoa e implementa as demais características:

**A classe Pessoa:**

```
1  package interfaces;
2
3  import java.util.Date;
4
5  public class Pessoa {
6
7      public String nome;
8      public String endereco;
9      public Date dataNacimento;
10
11     public void andar() {}
12 }
```

**A Interface Atleta:**

```
1  package interfaces;
2
3  public interface Atleta {
4
5      public void aquecer();
6      public void alongar();
7      public void competir();
8  }
```

**A Interface Ciclistas:**

```
1  package interfaces;
2
3  public interface Ciclistas extends Atleta {
4
5      public void correrDeBicicleta();
6  }
```

**A Interface Corredores:**

```
1  package interfaces;
2
3  public interface Corredores extends Atleta {
4
5      public void correr();
6  }
```

**A Interface Nadadores:**

```
1  package interfaces;
2
3  public interface Nadadores extends Atleta {
4
5      public void nadar();
6  }
```

**A classe Triatleta:**

```
 1 package interfaces;
 2
 3 public class Triatleta extends Pessoa implements Ciclistas, Nadadores,
 4         Corredores {
 5
 6     public Triatleta(String nome) {
 7         this.nome = nome;
 8     }
 9
10     public void correrDeBicicleta() {
11         System.out.println(this.nome + " esta correndo de bicicleta!");
12     }
13
14     public void nadar() {
15         System.out.println(this.nome + " esta nadando!");
16     }
17
18     public void correr() {
19         System.out.println(this.nome + " esta correndo!");
20     }
21
22     public void aquecer() {
23         System.out.println(this.nome + "esta aquecendo!");
24     }
25
26     public void alongar() {
27         System.out.println(this.nome + "esta alongando");
28     }
29
30     public void competir() {
31         System.out.println(this.nome + "esta competindo!");
32     }
33
34 }
```

**A classe TestaInterfaces:**

```
 1 package interfaces;
 2
 3 public class TestaInterfaces {
 4
 5     public static void main(String[] args) {
 6         Triatleta atleta = new Triatleta("Fernanda Keller");
 7         atleta.nadar();
 8         atleta.correrDeBicicleta();
 9         atleta.correr();
10     }
11 }
```

**Classes internas:**

Uma classe interna, nada mais é do que uma classe que é definida de dentro de outra. Mas, por que alguém ia querer fazer isso? Existem alguns motivos para isso:

- Um objeto de uma classe interna pode acessar a implementação do objeto que a criou – incluindo dados que seriam, de outra forma, privados.
- As classes internas podem ser invisíveis para outras classes do mesmo pacote.
- As classes internas (principalmente classes anônimas) são muito convenientes quando do tratamento de eventos.

```
1  package classes;
2
3  public class Empregado {
4
5      private String nome;
6      private String sexo;
7      private float salario;
8      private HorarioTrabalho horarios;
9
10     public Empregado(String n, String s, float sal,
11             String entrada, String saida) {
12         nome = n;
13         sexo = s;
14         salario = sal;
15         horarios = new HorarioTrabalho(entrada, saida);
16     }
17
18     public void aumentarSalario(float percentual) {
19         salario = salario * (1 + percentual / 100);
20     }
21
22     public void print() {
23         System.out.println("Nome....: " + nome);
24         System.out.println("Sexo....: " + sexo);
25         System.out.println("Salario.: " + salario);
26         System.out.println("Entrada.: " + horarios.horaEntrada);
27         System.out.println("Saida...: " + horarios.horaSaida);
28     }
29
30     class HorarioTrabalho {
31
32         public String horaEntrada;
33         public String horaSaida;
34
35         public HorarioTrabalho(String horaEntrada, String horaSaida) {
36             this.horaEntrada = horaEntrada;
37             this.horaSaida = horaSaida;
38         }
39
40     }
41 }
```

**Tratamento de eventos e classes internas anônimas:**

A manipulação de eventos é de importância fundamental em programas GUI. Qualquer SO que suporte interfaces de usuário gráficas precisa constantemente monitorar o ambiente buscando por eventos tais como teclas pressionadas ou cliques do mouse. Cada programa decide então o que fazer em resposta a esses eventos. Em linguagens como Visual Basic e Delphi a correspondência entre eventos e código é obvia. O programador escreve o código para cada evento específico de seu interesse e coloca o código no que é, geralmente, chamado de manipulador de evento (ou procedimento de evento). Por outro lado se você usar uma linguagem "crua" como C para elaborar uma programação dirigida por eventos, terá que escrever um código para verificar continuamente a fila de eventos do sistema operacional. (Isso geralmente é feito colocando o código em um laço gigante através de um instrução switch monstruosa).

A linguagem Java adota uma metodologia entre esses dois extremos em termos de recursos e, conseqüentemente, na complexidade resultante. Dentro dos limites dos eventos que o AWT conhece, pode-se controlar totalmente como os eventos são transmitidos desde as origens de eventos (como botões e barra de rolagens) até os ouvintes de eventos. Pode-se designar qualquer objeto para ser um ouvinte de evento. As origens de eventos possuem métodos que permitem registrar os ouvintes de eventos neles. Quando um evento ocorre na origem, esta envia uma notificação desse evento para todos os objetos ouvintes que foram registrados para esse evento. Como era de se esperar de uma linguagem orientada a objetos como Java, a informação sobre o evento é encapsulada em um objeto evento. Em Java, todos os objetos evento, no fim, derivam da classe java.util.EventObject. Evidentemente, há subclasses para cada ripo de evento, como ActionListener e WindowEvent.

Origens de eventos diferentes podem produzir tipos diferentes de eventos. Por exemplo, um botão pode enviar objetos ActionEvent, enquanto uma janela pode enviar objetos WindowEvent.

Em resumo:

- Um objeto ouvinte é uma instância de uma classe que implementa uma Interface especial chamada, naturalmente, de Interface ouvinte (listener).
- Uma origem do evento é um objeto que pode registrar objetos ouvintes e envia a esses os objetos evento.
- A origem do evento envia objetos eventos para todos os ouvintes registrados quando esse evento ocorre.
- Os objetos ouvintes vão usar informação do objeto evento recebido para determinar a sua reação ao evento.

Exemplo:

```java
package gui;

import java.awt.*;
import java.awt.event.*;

import javax.swing.*;

public class TestaClickBotao extends JFrame implements ActionListener {

    private JButton botaoAzul    = new JButton("Azul");
    private JButton botaoAmarelo = new JButton("Amarelo");
    private JPanel  painel       = new JPanel();


    public TestaClickBotao() {
        setSize(200,80);

        botaoAzul.addActionListener(this);
        botaoAmarelo.addActionListener(this);
        painel.add(botaoAzul);
        painel.add(botaoAmarelo);
        getContentPane().add(painel);
    }

    public static void main(String[] args) {
        TestaClickBotao teste = new TestaClickBotao();
        teste.show();
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == botaoAzul) {
            this.painel.setBackground(Color.BLUE);
        }
        if (e.getSource() == botaoAmarelo) {
            this.painel.setBackground(Color.YELLOW);
        }
    }
}
```

**Capturando eventos de janelas:**

Por padrão, quando clicamos no botão (X) de uma janela, ela não fecha, mas fica oculta. Para podermos realmente fechar a janela, necessitamos que os eventos dessa janela sejam capturados. Quando o usuário tenta fechar a janela, um evento WindowEvent é gerado pela classe JFrame. Para capturar esse evento, precisamos de um obejto ouvinte e registrá-los à lista de ouvintes da janela:

```
1  package gui;
2
3  import java.awt.event.*;
4
5  public class FechaFrame extends WindowAdapter {
6
7      public void windowClosing(WindowEvent e) {
8          System.exit(0);
9      }
10 }
```

Além de criarmos essa classe, precisamos adicionar a seguinte linha na classe da janela que queremos fechar: (nesse caso, na classe TestaClickBotao).

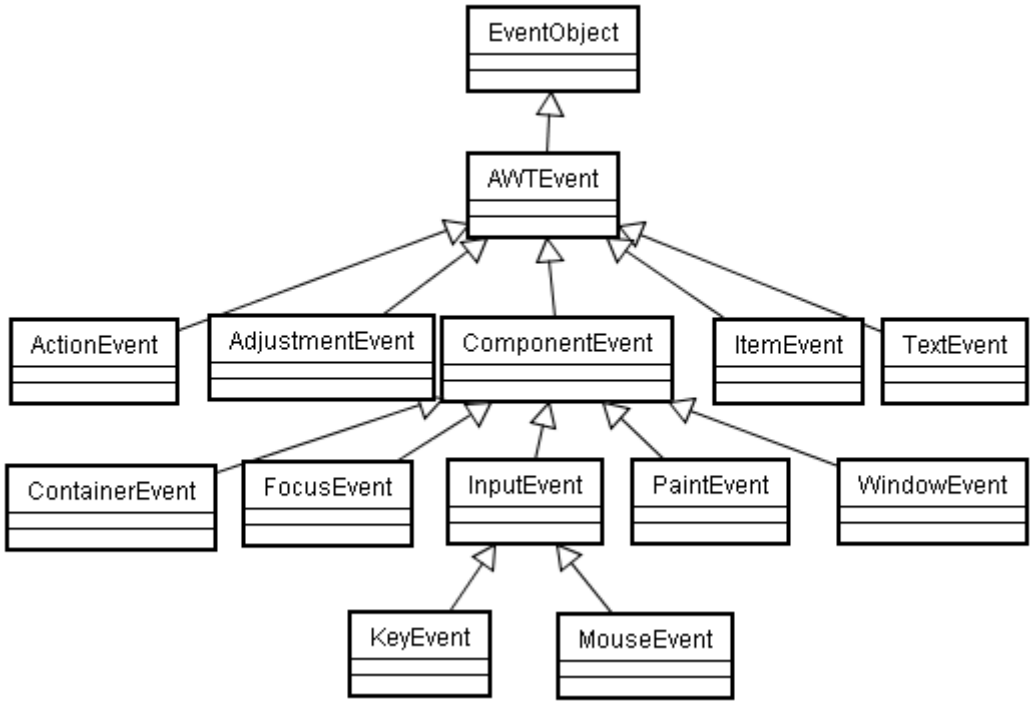
```
18          botaoAzul.addActionListener(this);
19          botaoAmarelo.addActionListener(this);
20          painel.add(botaoAzul);
21          painel.add(botaoAmarelo);
22          getContentPane().add(painel);
23
24          FechaFrame fecha = new FechaFrame();
25          this.addWindowListener(fecha);
26
```

Obs.: A classe WindowAdapter é uma classe adaptadora. As classes adptadoras existem para evitar que o programador digite código "desnecessário". Como já sabemos todas as vezes que implementamos uma Interface, obrigatoriamente temos que implementar todos os seus métodos. A Interface responsável pelos eventos de janela é WindowListener. Essa interface possui 7 métodos, ou seja, se você implementá-la diretamente na sua classe terá que implementar seus 7 métodos. Porém estamos interessados somente em um desses métodos: windowClosing. E teríamos que implementar os outros sem nenhum código. Para evitar isso, a classe adaptadora WindowAdapter já implementa a interface WindowEvent e seus 7 métodos. Com isso, quando estendemos dessa classe, precisamos implementar somente os métodos que estamos interessados.

**Métodos da Interface WindowListener:**

| |
|---|
| public void windowClosed(WindowEvent e) |
| public void windowIconified(WindowEvent e) |
| public void windowOpened(WindowWvent e) |
| public void windowClosing(WindowEvent e) |
| public void windowDeiconified(WindowEvent e) |
| public void windowActivated(WindowEvent e) |
| public void windowDeactivated(WindowEvent e) |

**Hierarquia de eventos do AWT:**




Algumas das classes de eventos do AWT, não tem uso prático para o programador. Por exemplo, o AWT insere objetos PaintEvent na fila de eventos, mas esses objetos não se destinam aos ouvintes. Eis uma lista de tipos de eventos AWT que realmente são destinados aos ouvintes:

| | |
|---|---|
| ActionEvent | ItemEvent |
| AdjustmentEvent | KeyEvent |
| ComponentEvent | MouseEvent |
| ContainerEvent | TextEvent |
| FocusEvent | WindowEvent |

O pacote javax.swing.event contém eventos adicionais que são específicos dos componentes Swing. Há onze interfaces ouvintes reunidas no pacote java.awt.event:

| | |
|---|---|
| ActionListener | KeyListener |
| AdjustmentListener | MouseListener |
| ComponentListener | MouseMotionListener |
| ContainerListener | TextListener |
| FocusListener | WindowListener |
| ItemListener | |

**Tabela com o resumo sobre a manipulação de eventos AWT:**

| Interface | Métodos | Parâmetro | Gerados por |
|---|---|---|---|
| ActionListener | actionPerformed | ActionEvent | Button<br>List<br>MenuItem<br>TextField |
| AdjustmentListener | adjustmentValueChanged | ItemEvent | ScrollBar |
| ItemListener | itemStateChanged | ItemEvent | Checkbox<br>CheckboxMenuItem<br>Choice<br>List |
| TextListener | textValueChanged | TextEvent | TextComponent |
| ComponentListener | componentMoved<br>componentHidden<br>componentResized<br>componentShow | ComponentEvent | Component |
| ContainerListener | componentAdded<br>componentRemoved | ContainerEvent | Container |
| FocusListener | focusGained<br>focusLost | FocusEvent | Component |
| KeyListener | keyPressed<br>keyReleased<br>keyTyped | KeyEvent | Component |
| MouseListener | mousePressed<br>mouseReleased<br>mouseEntered<br>mouseExited<br>mouseClicked | MouseEvent | Component |
| MouseMotionListener | mouseDragged<br>mouseMoved | MouseEvent | Component |
| WindowListener | windowClosing<br>windowOpened<br>windowIconified<br>windowDeiconified<br>windowClosed<br>windowActivated<br>windowDeactivated | WindowEvent | Window |

**Classes internas anônimas:**

As vezes queremos construir apenas um único objeto de uma classe. Quando isso acontece, podemos utilizar uma classe anônima. È chamada assim, por não é necessário dar-lhe um nome.

Vamos ver como ficaria o exemplo do click dos botões utilizando classes anônimas para o tratamento dos eventos:

```java
package gui;

import java.awt.*;
import java.awt.event.*;

import javax.swing.*;

public class TestaClickBotao1 extends JFrame {

    private JButton botaoAzul    = new JButton("Azul");
    private JButton botaoAmarelo = new JButton("Amarelo");
    private JPanel  painel       = new JPanel();

    public TestaClickBotao1() {
        setSize(200,80);

        botaoAzul.addActionListener(

            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    painel.setBackground(Color.BLUE);
                }
            }
        );

        botaoAmarelo.addActionListener(

            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    painel.setBackground(Color.YELLOW);
                }
            }
        );

        painel.add(botaoAzul);
        painel.add(botaoAmarelo);
        getContentPane().add(painel);

        FechaFrame fecha = new FechaFrame();
        this.addWindowListener(fecha);

    }

    public static void main(String[] args) {
        TestaClickBotao teste = new TestaClickBotao();
        teste.show();
    }
}
```