

# Programação Orientada por Objetos

---

## **JavaFX – Propriedades**

Prof. José Cordeiro,

Prof. Cédric Grueau,

Prof. Laercio Júnior

Departamento de Sistemas e Informática

Escola Superior de Tecnologia de Setúbal – Instituto Politécnico de Setúbal

2019/2020

- Sessão 1 – Visualizador de Pessoas
  - Criação de uma interface gráfica
  - Especializar (herança) objetos gráficos para representar a nossa informação
  
- Sessão 2 – Propriedades e *Binding*



Módulo 12 – JavaFX – Propriedades

# **SESSÃO 1 – EXEMPLO VISUALIZADOR PESSOAS**

# Exemplo — Visualizador de Pessoas

- Requisitos da aplicação:
  - Criar uma aplicação que permita gerir a informação de um grupo de pessoas.
  - A informação a guardar para cada pessoa é apenas o nome e a idade.
  - Deve ser possível adicionar, alterar, visualizar e apagar uma pessoa.
  - A visualização deve ser feita em JavaFX em duas cenas diferentes, uma que mostra as pessoas e outra que visualiza apenas uma pessoa.



# Criação de uma interface gráfica

- ❑ Supor que temos uma classe **Pessoa**, que representa a informação associada a uma pessoa:
  - ❑ Nome
  - ❑ Idade
- ❑ Definir a classe com a garantia de não haver informação inconsistente:
  - ❑ Existe o nome
  - ❑ A idade é [0,120]
- ❑ Em caso de valores impossíveis o construtor lança exceções

```
public class Pessoa {  
  
    private static final int IDADE_MINIMA = 0;  
    private static final int IDADE_MAXIMA = 120;  
    private String nome;  
    private int idade;  
  
    protected static boolean nomeValido(String nome) {  
        return (nome != null && !nome.isEmpty());  
    }  
  
    protected static boolean idadeValida(int idade) {  
        return ((idade >= IDADE_MINIMA)  
            && (idade <= IDADE_MAXIMA));  
    }  
  
    public Pessoa(String nome, int idade) {  
        if (!nomeValido(nome)) {  
            throw new IllegalArgumentException("O nome da  
pessoa não é válido!");  
        }  
        if (!idadeValida(idade)) {  
            throw new IllegalArgumentException("A idade da  
pessoa não é válida!");  
        }  
        this.nome = nome;  
        this.idade = idade;  
    }  
    // Continua...
```

# Classe Pessoa (cont.)

- ❑ Tradicionais **getters** (inspetores)
- ❑ Nos **setters** (modificadores) é validado o valor antes da alteração
- ❑ Criar *getters* para as constantes, para serem acessíveis externamente
- ❑ Convém redefinir o **toString**

```
public String getNome() {  
    return nome;  
}  
  
public void setNome(String nome) {  
    if (nomeValido(nome)) {  
        this.nome = nome;  
    }  
}  
  
public int getIdade() {  
    return idade;  
}  
  
public void setIdade(int idade) {  
    if (idadeValida(idade)) {  
        this.idade = idade;  
    }  
}  
  
public static int getIdadeMinima() {  
    return IDADE_MINIMA;  
}  
  
public static int getIdadeMaxima() {  
    return IDADE_MAXIMA;  
}  
  
@Override  
public String toString() {  
    return getNome() + " (" + getIdade() + " anos)";  
}  
}
```

# Criar uma coleção de Pessoas

- ❑ Criar uma classe para armazenar diversas pessoas
- ❑ Optar por um conjunto pois não devem existir pessoas repetidas
- ❑ Implementar os modificadores (adicionar e remover) e os inspetores (verificar se está contido e obter, num array, todos os elementos)

```
public class Pessoas {  
  
    private HashSet<Pessoa> pessoas;  
  
    public Pessoas() {  
        pessoas = new HashSet<>();  
    }  
  
    public void adicionarPessoa(Pessoa pessoa) {  
        pessoas.add(pessoa);  
    }  
  
    public void removerPessoa(Pessoa pessoa) {  
        pessoas.remove(pessoa);  
    }  
  
    public boolean contemPessoa(Pessoa pessoa) {  
        return pessoas.contains(pessoa);  
    }  
  
    public Pessoa[] getPessoas() {  
        return pessoas.toArray(new Pessoa[pessoas.size()]);  
    }  
}
```

# Criar uma coleção de Pessoas - Alternativa

- ❑ Caso não seja necessária nenhuma ação adicional nos métodos que acessam à coleção (podem sempre ser **Override**).
- ❑ E caso não seja contraindicado utilizar os nomes originais dos métodos da coleção.
- ❑ É possível definir a classe **Pessoas** como uma extensão de **HashSet<Pessoa>**.
- ❑ Os métodos **adicionarPessoa**, **removerPessoa** e **contemPessoa**, passam a ser os **add**, **remove** e **contains** herdados da superclasse **HashSet<Pessoa>**.

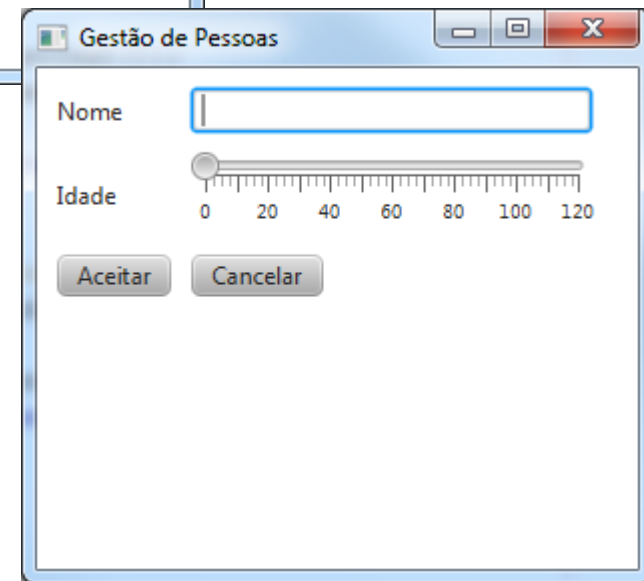
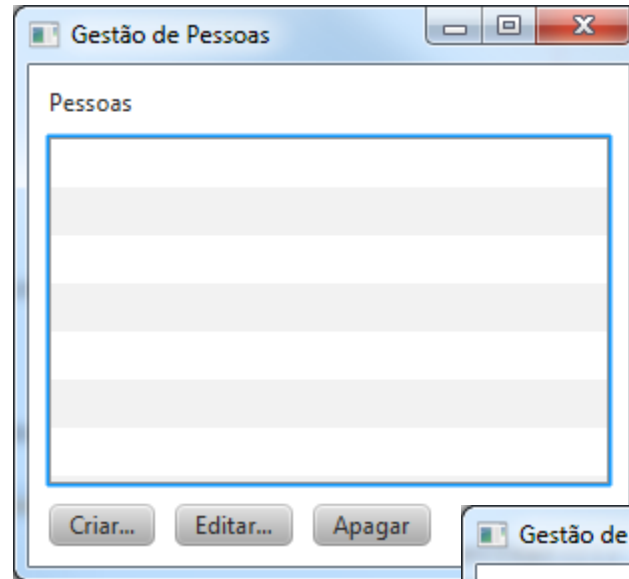
```
public class Pessoas extends HashSet<Pessoa> {  
  
}
```

```
public class Pessoas extends HashSet<Pessoa> {  
  
    @Override  
    public boolean add(Pessoa pessoa) {  
        System.out.println("Adicionei: " + pessoa);  
        return super.add(pessoa);  
    }  
  
    @Override  
    public boolean remove(Object pessoa) {  
        System.out.println("Removi: " + pessoa);  
        return super.remove(pessoa);  
    }  
}
```



# Definir a interface gráfica

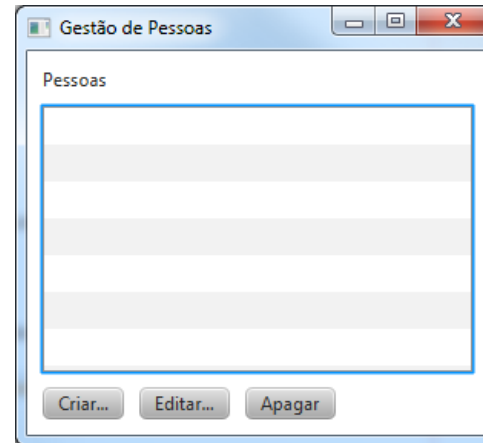
- ❑ Será necessário ter dois ecrãs:
  - ❑ Visualização e alteração da coleção
  - ❑ Visualização, criação e alteração dos dados de uma pessoa
- ❑ O utilizador controla a troca de ecrãs através dos botões Criar/Editar e Aceitar/Cancelar
- ❑ Serão criadas subclasses de **VBox** e de **GridPane** para devolverem os painéis prontos, com todos os elementos gráficos necessários
- ❑ Os construtores recebem os objetos com a informação a apresentar nos elementos gráficos



# Visualizador da coleção Pessoas

- É criada uma subclasse de **VBox** para apresentar a coleção de pessoas. O painel apresentará:

- O rótulo
- A lista
- Os botões



```
public class VisualizadorPessoas extends VBox {  
  
    public VisualizadorPessoas(final Pessoas pessoas) {  
        //Criar o rótulo  
        Label labelLista = new Label("Pessoas");  
  
        //Criar a lista  
        final ObservableList<Pessoa> listaPessoas = FXCollections.observableArrayList(pessoas);  
        final ListView<Pessoa> lista = new ListView<>(listaPessoas);  
    }  
}
```

# Visualizador da coleção Pessoas (cont.)

```
//Criar o painel dos botões
HBox painelBotoes = new HBox(10);

//Botão criar
Button botaoCriar = new Button("Criar...");
botaoCriar.setOnAction (e -> lista.getScene().setRoot(new VisualizadorPessoa(pessoas, null)));

//Botão editar
Button botaoEditar = new Button("Editar...");
botaoEditar.setOnAction (e -> {
    Pessoa pessoa = lista.getSelectionModel().getSelectedItem();
    if (pessoa != null) {
        lista.getScene().setRoot(new VisualizadorPessoa(pessoas, pessoa));
    }
});

//Botão apagar
Button botaoApagar = new Button("Apagar");
botaoApagar.setOnAction (e -> {
    Pessoa pessoa = lista.getSelectionModel().getSelectedItem();
    if (pessoa != null) {
        listaPessoas.remove(pessoa);
        pessoas.remove(pessoa);
    }
});

//Adicionar os botões
painelBotoes.getChildren ().addAll(botaoCriar, botaoEditar, botaoApagar);
```

# Visualizador da coleção Pessoas (cont.)

- ❑ Depois de criados todos os elementos, estes devem ser inseridos no respetivo painel com espaçamentos definidos (Padding e Spacing).

```
//Inserir os nós folha no nó ramo

setPadding( new Insets(10));
setSpacing(10);
getChildren().setAll(labelLista, lista, painelBotoes);
}
```

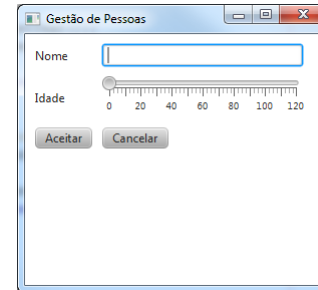
- ❑ Nesta classe só se definiu:
  - ❑ O construtor para inserir os elementos gráficos necessários
  - ❑ Os **EventHandler** dos botões que manipulam a coleção (argumento do construtor – final)Todas as restantes funcionalidades são herdadas do painel **VBox**
- ❑ A alternância entre ecrãs é feita através de **xxx.getScene().setRoot()**, onde **xxx** representa um elemento gráfico colocado no painel. Assim ter-se-á acesso à **scene** onde o painel está a ser apresentado.

# Visualizador de Pessoa

- É criada **uma subclasse** de **GridPane** para apresentar uma **Pessoa**.

O painel apresentará:

- Os rótulos
- A caixa de texto para o nome
- O *slider* para a idade
- Os botões



```
public class VisualizadorPessoa extends GridPane {  
  
    public VisualizadorPessoa(final Pessoas pessoas, final Pessoa pessoa) {  
        //Criar o rótulo do nome  
        Label nomeRotulo = new Label("Nome");  
        //Criar a caixa de texto para o nome  
        final TextField nomeCampo = new TextField();  
        nomeCampo.setPrefWidth(200);  
        //Criar o rótulo da idade  
        Label idadeRotulo = new Label("Idade");  
        //Criar o slider da idade  
        final Slider idadeSlider = criarSliderIdade();  
        //Inicializar com a informação da pessoa  
        if (pessoa != null) {  
            nomeCampo.setText(pessoa.getNome());  
            idadeSlider.setValue(pessoa.getIdade());  
        }  
    }  
}
```

# Visualizador de Pessoa — criação do Slider

- A criação do **Slider** deve utilizar os limites da idade definidos na classe **Pessoa**, acrescentamos, à classe atual (**VisualizadorPessoa**), um método que cria e devolve um **Slider**, adaptado à informação das idades das pessoas:

```
public class VisualizadorPessoa extends GridPane {  
  
    ...  
  
    public final Slider criarSliderIdade() {  
        Slider slider = new Slider();  
        slider.setMin(Pessoa.getIdadeMinima());  
        slider.setMax(Pessoa.getIdadeMaxima());  
        slider.setShowTickLabels(true);  
        slider.setShowTickMarks(true);  
        slider.setMajorTickUnit(10);  
        return slider;  
    }  
  
    ...  
}
```

# Criação do Slider - nova classe

- Alternativamente a criação do **Slider** poderia ser feita através da especialização da classe **Slider**, criando-se um tipo de **Slider** específico para as idades. Basta copiar o conteúdo do método anterior para o construtor dessa nova classe (**SliderIdade**):

```
public class SliderIdade extends Slider {  
  
    public SliderIdade() {  
        setMin(Pessoa.getIdadeMinima());  
        setMax(Pessoa.getIdadeMaxima());  
        setShowTickLabels(true);  
        setShowTickMarks(true);  
        setMajorTickUnit(10);  
    }  
}
```

## Criação do Slider - nova classe (cont.)

- Neste segundo caso, será necessário substituir a criação do **Slider** pela criação de um objeto da nova classe (**SliderIdade**) na classe **VisualizadorPessoa**,

```
public class VisualizadorPessoa extends GridPane {  
  
    public VisualizadorPessoa(final Pessoas pessoas, final Pessoa pessoa) {  
        ...  
        final Slider idadeSlider = criarSliderIdade();  
        final SliderIdade idadeSlider = new SliderIdade();  
        ...  
    }  
}
```

- O método anterior, **criarSliderIdade()**, da classe **VisualizadorPessoa** deve ser eliminado.



## Visualizador de Pessoa (cont.)

- ❑ Depois de criados os elementos para apresentar a informação da pessoa convém acrescentar à janela **VisualizadorPessoa** os botões aceitar e cancelar:

```
//Botão aceitar
Button botaoAceitar = new Button("Aceitar");
botaoAceitar.setOnAction(e -> {
    if (pessoa == null) { //Criar
        try {
            pessoas.add(new Pessoa(nomeCampo.getText(), (int) idadeSlider.getValue()));
        } catch (Exception ex) {
        }
    } else { //Atualizar
        pessoa.setNome(nomeCampo.getText());
        pessoa.setIdade((int) idadeSlider.getValue());
    }
    nomeCampo.getScene().setRoot(new VisualizadorPessoas(pessoas));
});
```


```
//Botão cancelar
Button botaoCancelar = new Button("Cancelar");
botaoCancelar.setOnAction(e ->
    nomeCampo.getScene().setRoot(new VisualizadorPessoas(pessoas)));
```

# Visualizador de Pessoa (cont.)

- No final da criação dos botões são colocados todos os elementos gráficos:

```
//Posicionar os nós
setPadding(new Insets(10));
setVgap(10);
setHgap(10);
add(nomeRotulo, 0, 0);
add(nomeCampo, 1, 0);
add(idadeRotulo, 0, 1);
add(idadeSlider, 1, 1);
add(botaoAceitar, 0, 2);
add(botaoCancelar, 1, 2);
    }
}
```

# Gravar a informação

- ❑ Com esta interface é possível ir acrescentando, modificando e apagando a informação associada às pessoas.
- ❑ A manutenção da informação entre execuções do programa passa por gravar, num ficheiro, toda a informação
- ❑ É preciso tornar **Serializable** as classes (**Pessoas** e **Pessoa**) e definir métodos, **lerPessoas()** e **gravarPessoas()**, para ler (no início do programa) e escrever (associado ao botão fechar  ) a informação de/para o ficheiro:

```
@Override
public void start(Stage primaryStage) {
    final Pessoas pessoas = Pessoas.lerPessoas();
    Scene scene = new Scene(new VisualizadorPessoas(pessoas), 400, 300);
    primaryStage.setTitle("Gestão de Pessoas");
    primaryStage.setScene(scene);
    primaryStage.setOnCloseRequest(e -> pessoas.gravarPessoas());
    primaryStage.show();
}
```

# Gravar a informação

```
public class Pessoas extends HashSet<Pessoa> implements Serializable {

    private static final String NOME_FICHEIRO = "pessoas.dat";

    ...

    public void gravarPessoas() {
        try {
            ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(NOME_FICHEIRO));
            oos.writeObject(this);
            oos.flush();
            oos.close();
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }

    public static Pessoas lerPessoas() {
        Pessoas pessoas;
        try {
            ObjectInputStream ois = new ObjectInputStream(new FileInputStream(NOME_FICHEIRO));
            pessoas = (Pessoas) ois.readObject();
            ois.close();
        } catch (IOException | ClassNotFoundException e) {
            System.out.println(e.getMessage());
            pessoas = new Pessoas();
        }
        return pessoas;
    }
}
```



Módulo 12 – JavaFX – Propriedades

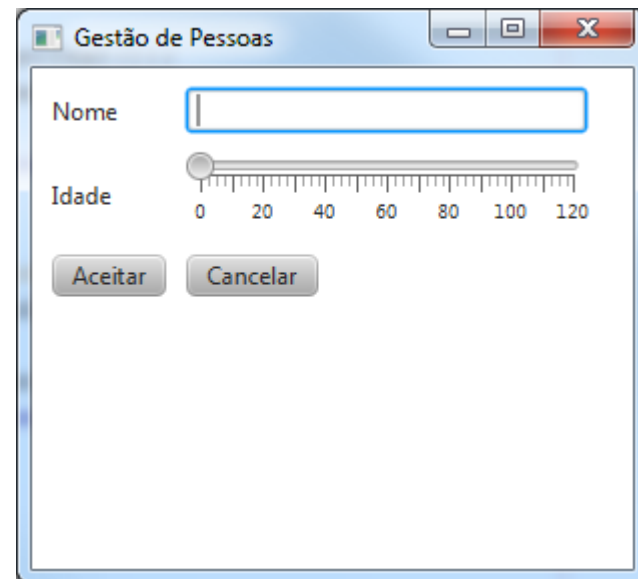
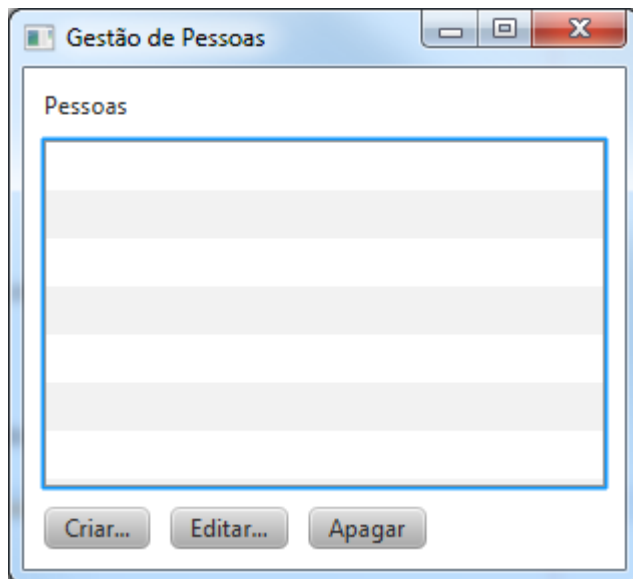
## **SESSÃO 2 – PROPRIEDADES**

# Exemplo — Visualizador de Pessoas

- Requisitos da aplicação:
  - Criar uma aplicação que permita gerir a informação de um grupo de pessoas.
  - A informação a guardar para cada pessoa é apenas o nome e a idade.
  - Deve ser possível adicionar, alterar, visualizar e apagar uma pessoa.
  - A visualização deve ser feita em JavaFX em duas cenas diferentes, uma que mostra as pessoas e outra que visualiza apenas uma pessoa.

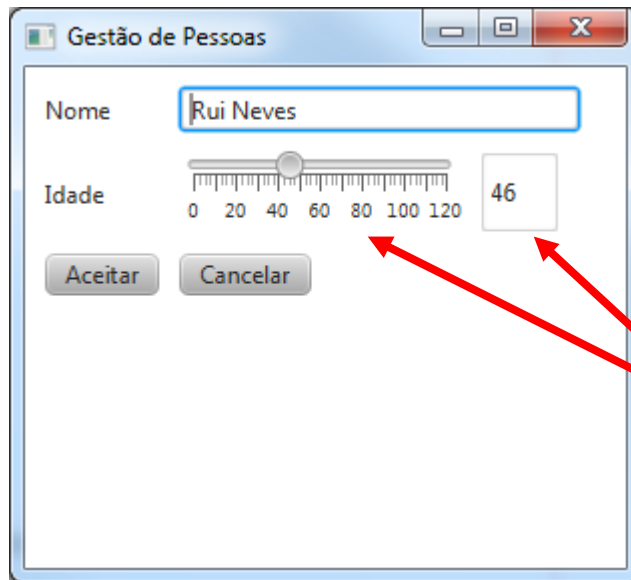


# Exemplo — Visualizador de Pessoas



# Sincronização da informação

- ❑ A escolha da introdução da informação relativa à idade através de um **Slider** é visualmente apelativa mas não permite a introdução de informação de forma precisa,
- ❑ O ideal era ter a possibilidade de introdução da informação através do **Slider** e de uma Caixa de Texto, de forma sincronizada:



A informação introduzida no **Slider** e na Caixa de Texto está sincronizada: mudando um, muda o outro



# Sincronização da informação

- ❑ O **Slider** tem o *getter* e o *setter* tradicionais: **getValue()** e **setValue()**. No entanto o valor é "armazenado" dentro de um objeto especial que se obtém através de **valueProperty()** e não num simples **double**.
- ❑ Analogamente, a Caixa de Texto tem o *getter* e o *setter* tradicionais: **getText()** e **setText()**, mas o seu valor é "armazenado" dentro de um objeto especial que se obtém através de **textProperty()** e não numa simples **String**.
- ❑ Estes objetos especiais, as "**propriedades**", permitem associações entre eles, por forma a sincronizar a sua informação. Esta sincronização pode ser bidirecional (a alteração de um implica, automaticamente, a alteração do outro) ou unidirecional (só a mudança de um é que influencia a mudança do outro).
- ❑ A sincronização bidirecional é feita através do método **bindBidirectional** e a unidirecional através de **bind**.
- ❑ É possível desfazer a sincronização automática através dos métodos **unbindBidirectional** e **unbind**, respetivamente.

# Propriedades

- ❑ Existem **propriedades** para armazenar quase todos os tipos de valores primitivos:
  - ❑ **BooleanProperty, DoubleProperty, FloatProperty, IntegerProperty, LongProperty**
  - ❑ Existem ainda as propriedades **ObjectProperty** e **StringProperty**, para armazenar objetos genéricos e **Strings**
- ❑ As anteriores classes são abstratas e devem ser especializadas para implementar funcionalidades específicas. No entanto estão disponíveis para instanciação as seguintes classes concretas:
  - ❑ **SimpleBooleanProperty, SimpleDoubleProperty, SimpleFloatProperty, SimpleIntegerProperty, SimpleLongProperty, SimpleObjectProperty, SimpleStringProperty**
- ❑ Podemos modificar os tipos dos atributos das nossas classes, alterando os seus tipos primitivos, para as propriedades equivalentes:
  - ❑ ex.: de **private int idade** para **private IntegerProperty idade**) e obter atributos sincronizáveis (usar **get()** e **set()**).

# Propriedades — Listener

- ❑ As propriedades permitem, para além da sincronização, associar um **Listener**, que será executado sempre que o valor da propriedade é alterado.
- ❑ Desta forma podemos implementar comportamentos genéricos, que excedem a simples sincronização do valor. Por exemplo, para uma propriedade **DoubleProperty** (exemplo com o **Slider** da idade):

```
idadeSlider.valueProperty().addListener(new ChangeListener<Number>() {  
    @Override  
    public void changed(ObservableValue<? extends Number> observable,  
                        Number oldValue,  
                        Number newValue) {  
        "Código a executar quando a propriedade muda de valor."  
        Temos acesso:  
            à propriedade (observable),  
            ao valor original (oldValue)  
            ao novo valor (newValue)"  
    }  
});
```

# Propriedades no JavaFX

- ❑ As propriedades são um mecanismo fundamental do JavaFX.
- ❑ Todos os objetos gráficos têm propriedades para os seus diversos valores. O teste **hasProperties()** verifica se um **Node** tem ou não propriedades.
- ❑ As propriedades são acessíveis através de métodos da forma **xxxProperty()**.
- ❑ As propriedades são manipuladas através de **get()** e **set()**.
- ❑ A sincronização de propriedades vai ter um papel preponderante na implementação de animações no JavaFX, pois a sincronização de valores permitirá que estes (posição, dimensão, outras características, etc.) mudem ao longo do tempo da animação.

# Slider e Caixa de Texto

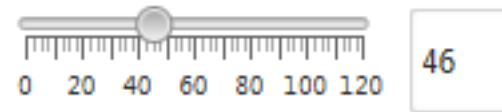
- ❑ Para resolver o problema de implementar um **Slider** sincronizado com uma Caixa de Texto, vai ser necessário criar uma classe que substitua a classe **Slider**, pelo menos nas suas funcionalidades básicas utilizadas no presente projeto (aceder através de **getValue()** e **setValue()**).
- ❑ Assim, na classe **VisualizadorPessoa**, substitui-se a criação do **SliderIdade** pela criação de um objeto da nova classe (**VisualizadorIdade**):

```
public class VisualizadorPessoa extends GridPane {  
  
    public VisualizadorPessoa(final Pessoas pessoas, final Pessoa pessoa) {  
        ...  
        final SliderIdade idadeSlider = new SliderIdade();  
        final VisualizadorIdade idadeSlider = new VisualizadorIdade();  
        ...  
    }  
}
```

# Classe VisualizadorIdade

- A implementação da classe **VisualizadorIdade** apenas necessita dos atributos para conter os elementos gráficos, do construtor e do *getter* e *setter*:

```
public class VisualizadorIdade extends HBox {  
  
    private final SliderIdade idadeSlider;  
    private final TextField idadeCampo;  
  
    public VisualizadorIdade() {  
        idadeSlider = new SliderIdade();  
        idadeCampo = new TextField();  
        idadeCampo.setPrefWidth(50);  
        idadeCampo.textProperty().bindBidirectional(idadeSlider.valueProperty(),  
            NumberFormat.getIntegerInstance());  
        setSpacing(10);  
        getChildren().addAll(idadeSlider, idadeCampo);  
    }  
  
    public double getValue() {  
        return idadeSlider.getValue();  
    }  
  
    public void setValue(double idade) {  
        idadeSlider.setValue(idade);  
    }  
}
```



# Resumindo

- ❑ Criação de interfaces gráficas através do acrescento de novas classes que permitem a visualização dos objetos já existentes no domínio do problema.
- ❑ Para cada classe poderá ser criado um "visualizador" (implementa as operações básicas – CRUD: Create, Read, Update e Delete).
- ❑ A criação dos visualizadores é feita por herança de objetos gráficos já existentes. Uma vez que se pretende apresentar informação dos vários atributos, recorre-se, normalmente, à extensão das classes que permitem agrupar/apresentar diversos elementos gráficos – os painéis. No respetivo construtor (que recebe o objeto a apresentar) são criados todos os elementos gráficos necessários.
- ❑ A funcionalidade fica completa através da criação dos diversos **EventHandler** e da ligação (*binding*) entre as **propriedades** do objeto e a dos elementos gráficos. Esta ligação pode ser feita de forma bidirecional (**bindBidirectional**) ou unidirecional (**bind**), para sincronizar os valores ou genericamente através da criação de um **Listener**.

Chapter 3 – Lambdas and Properties Pgs 74 a 89

Chapter 4 – Layouts and UI Controls Pgs 91 a 101 e 108 a 111

Propriedades e binding:

<http://docs.oracle.com/javase/8/javafx/properties-binding-tutorial/binding.htm#sthref8>

