

JC2002 Java 程序设计

第 9 天：摇摆模型、并发（CS）

11 月 13 日星期一

JC2002 Java 程序设计

第 9 天，第 1 课时：摇摆中的模特

参考文献和学习目标

- 今天的前两节课主要以 Oracle 文档为基础：
 - <https://docs.oracle.com/javase/tutorial/uiswing>
- 课程结束后，您应该能够
 - 在 Java 图形用户界面实施中使用 Swing 模型
 - 在 JList 和 JTable 组件中实现自定义功能

摇摆模型

- 模型存储组件的状态（例如，助记符、是否启用、选择等）和数据（如列表中显示的项目）
 - 大多数 Swing 组件都有预定义的模式
 - 某些组件（如列表）有多种模式
 - 例如，JList 使用 ListModel 和 ListSelectionModel
- 对于简单的组件（如按钮），您通常会直接与组件交互，而对于更复杂的组件（如列表和表格），与模型交互是更好的选择

为什么要使用模型？

- 模型允许将数据与视图和
如果采用 MVC 模式，控制器
- 可对默认模型进行扩展，从而在决定如何存储和检索
数据时提供自定义功能和灵活性
- 模型会自动向所有已注册的监听器传播更改，从而更
新视图（即图形用户界面）

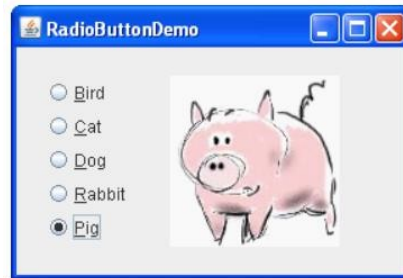
直接使用模型与组件

- 在 Java 中，有不同的方法可以实现相同的结果

```
JRadioButton pigButton = new JRadioButton("Pig");
pigButton.setMnemonic(KeyEvent.VK_P);
pigButton.setActionCommand("Pig");
pigButton.setSelected(true);

// 直接使用组件
System.out.println(pigButton.isSelected());

// 使用模型
DefaultButtonModel model = (DefaultButtonModel)pigButton.getModel();
System.out.println(model.isSelected());
```

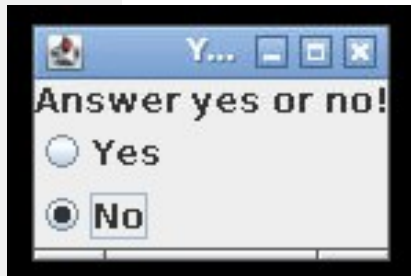


- 大多数从 JComponent 继承而来的组件类都有一个模型
默认情况下，可以使用方法 getModel()

直接与单选按钮互动

```
1  import javax.swing.*;
2  公共类 YesNoButtonExample {
3      public static void main(String[] args) {
4          JFrame frame = new JFrame("Yes or No?");
5          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
6          JPanel panel = new JPanel();
7          BoxLayout boxlayout = new BoxLayout(panel, BoxLayout.Y_AXIS);
8          panel.setLayout(boxlayout);
9          JLabel question = new JLabel("Answer yes or no!");
10         ButtonGroup group = new ButtonGroup();
11         JRadioButton yes = new JRadioButton("Yes");
12         JRadioButton no = new JRadioButton("No");
13         group.add(yes); group.add(no);
```

```
$ java YesNoButtonExample
Yes selected: false
没有选择: true
```

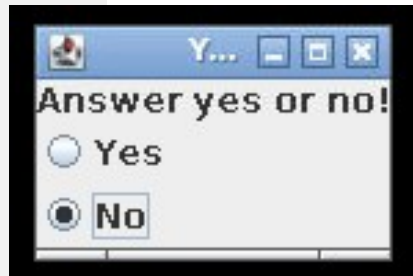



```
14     panel.add(question);
15     panel.add(yes); panel.add(no);
16     frame.add(panel);
17     frame.pack();
18     frame.setVisible(true);
19     no.setSelected(true);
20
21     System.out.println("Yes selected: "+yes.isSelected());
22
23     System.out.println("No selected: "+no.isSelected());
    }
```

通过模型与单选按钮互动

```
1  import javax.swing.*;
2  公共类 YesNoButtonExample2 {
3      public static void main(String[] args) {
4          JFrame frame = new JFrame("Yes or No?");
5          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
6          JPanel panel = new JPanel();
7          BoxLayout boxlayout = new BoxLayout(panel, BoxLayout.Y_AXIS);
8          panel.setLayout(boxlayout);
9          JLabel question = new JLabel("Answer yes or no!");
10         ButtonGroup group = new ButtonGroup();
11         JRadioButton yes = new JRadioButton("Yes");
12         JRadioButton no = new JRadioButton("No");
13         DefaultButtonModel yesModel = (DefaultButtonModel)yes.getModel();
14         ;
15         DefaultButtonModel noModel = (DefaultButtonModel)no.getModel();
```

```
15         ...d(no);
16
17         ...d(no);
18
```



没有选择: true

```
$ java YesNoButtonExample Yes selected: false
```

```
19      frame.pack();
20      frame.setVisible(true);
21      System.out.println("Yes selected: "+yesModel.isSelected());
22      System.out.println("No selected: "+noModel.isSelected());
23  }
```

定义自定义按钮模型

```
1  import javax.swing.*;
2
3  class CustomButtonModel extends JToggleButton.ToggleButtonModel
4  {
5      private AbstractButton button;
6      private String text;
7      CustomButtonModel(AbstractButton button) {
8          this.button = button;
9          text = button.getText();
10     }
11     public void printStatus() {
12         System.out.println(text + " 选中: " + isSelected());
13     }
14
15     @Override
16     public void setSelected(boolean b) {
17         if(b) {
18             button.setText(text + " (当前已启用)");
19         }
20         否则 {
21             button.setText(text + " (当前禁用)");
22         }
23
24         super.setSelected(b);
25     }
26 }
```

自定义单选按钮模型应继承切换按钮模型

增加功能的新方法

覆盖方法可实现额外功能

使用自定义按钮模型 (1)

```
1  import javax.swing.*;
2  类 CustomButtonModel 扩展 JToggleButton.ToggleButtonModel {
3      private AbstractButton button;
4      private String text;
5      客户
6      第 23 公共类 YesNoButtonExample2 {
7      茶 ... ..
8      }
9      2 发布 } 32
10     2  Sy 33
11     } } 34
12     @Ove 35
13     publ 36
14     如果 37
15     果 37
16     ... ..
17     } 45
18     el 46
19     } 47
20     su 48
```

49
50

```
J    o    no = new JRadioButton("No");
R    B
a    u
d    t
i    t
o    o
.. B    n
.    u    (
    t    "
    t    Y
    o    e
    n    s
    }    "
}    y    )
    e    ;
    s
    J
    =    R
    n    a
    e    d
    w    i
    J    o
    R    B
    a    u
    d    t
    i    t
    n    o
    n
```

```
no = new JRadioButton("No");
CustomButtonModel yesModel = new CustomButtonModel(yes);
CustomButtonModel noModel = new CustomButtonModel(no);
yes.setModel(yesModel);
no.setModel(noModel);
yes.setSelected(false);
no.setSelected(true);
yesModel.printStatus();
noModel.printStatus();
```

实例化自定义模型并分配给
单选按钮对象

使用自定义按钮模型 (2)

```
1  import javax.swing.*;
2  类 CustomButtonModel 扩展 JToggleButton.ToggleButtonModel {
3      private AbstractButton button;
4      private String text;
5      客户
6      第 23 公共类 YesNoButtonExample2 {
7      茶 ...    public static void main(String[] args) {
8  }
9      出版部门    JRadioButton yes = new JRadioButton("Yes");
10     32    JRadioButton no = new JRadioButton("No");
11     Sy 33    CustomButtonModel yesModel = new CustomButtonModel(yes);
12     } 34    CustomButtonModel noModel = new CustomButtonModel(no);
13     @Ove 35
14     发布 36    yes.setModel(yesModel);
15     如果 37    no.setModel(noModel);
16     } ...
17     el 45    yes.setSelected(false);
18     46    no.setSelected(true);
19     } 47    yesModel.printStatus();
```



```
20      苏 48  
21    } 49  
22  } 50  
    }
```

```
noModel.printStatus();
```

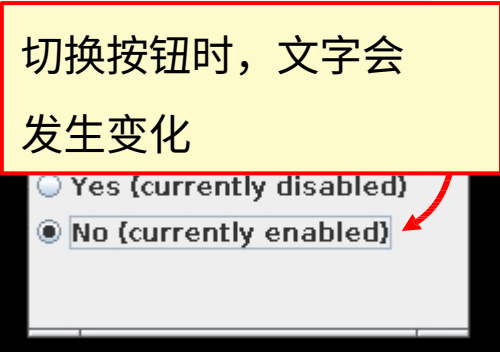
使用自定义方法
打印状态()

使用自定义按钮模型 (3)

```
1 import javax.swing.*;
2 类 CustomButtonModel 扩展 JToggleButton.ToggleButtonModel {
3     private AbstractButton button;
4     private String text;
5     客户
6     第 23 公共类 YesNoButtonExample2 {
7     茶 ... ..
8 }
9 出版部门
32 JRadioButton yes = new JRadioButton("Yes");
10 Sy 33 JRadioButton no = new JRadioButton("No");
11 } 34 CustomButtonModel yesModel = new CustomButt
12 @Ove 35 CustomButtonModel noModel = new CustomButto
13 发布 36 yes.setModel(yesModel);
14 如果 37 no.setModel(noModel);
15 } ... ..
16 } 45 yes.setSelected(false);
17 el 46 no.setSelected(true);
18 47 yesModel.printStatus();
19 } e4.7se
```

```
$ java YesNoButtonExample3
Yes_selected: false
没有选择: true
```

切换按钮时，文字会
发生变化



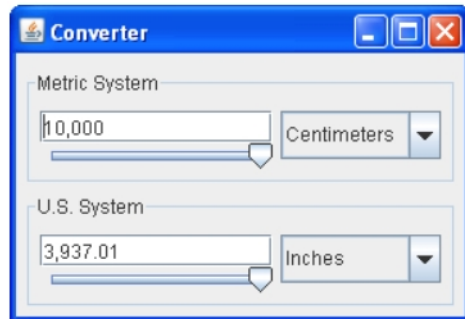
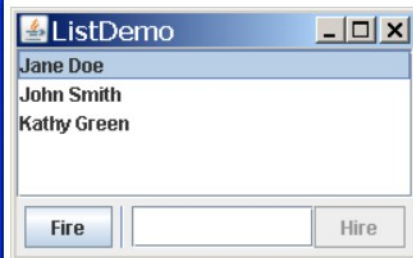
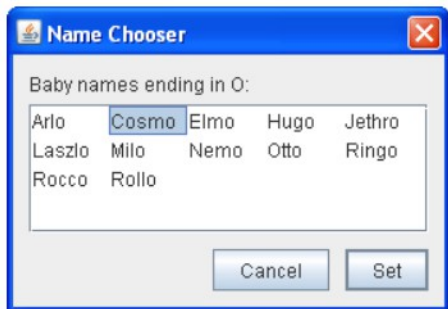
☐ Yes {currently disabled}

☒ No {currently enabled}

```
20      苏 48      noModel.printStatus();
21    }      49
22  }      50    }
```

利用模型进行复杂的互动

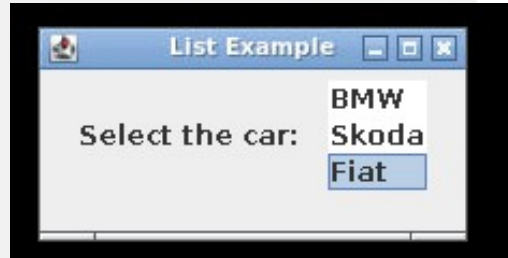
- 使用带有 JButton 等简单组件的模型的好处是通常是有限的，但对于复杂的组件，模型是必不可少的
- 通过 **JList** 和 **JTable** 等组件，模型可以实现更复杂的功能和交互
- 模型还有利于组件之间的互动



直接使用 JList 的简单示例

- JList 实例向用户展示一组项目，显示在一列或多列中，供用户选择

```
1 import java.awt.event.*;
2 import java.awt.*;
3 import javax.swing.*;
4
5 类 SimpleListExample {
6     public static void main(String[] args) {
7         JFrame frame = new JFrame("List Example");
8         JPanel panel = new JPanel();
9         JLabel label = new JLabel("Select the car: ");
10        String cars[] = {"BMW", "Skoda", "Fiat"};
11        Jlist<String> list = new Jlist(cars);
12        list.setSelectedIndex(2);
13    }
```



请注意，您需要定义 JList 中的项目（本例中为字符串）

```
panel.add(label);
```

```
panel.add(list);
```

直接在自己的类中使用 JList

- 您可以在 JList 中存储自己类的实例，但需要覆盖 toString() 方法，以控制项目的显示方式

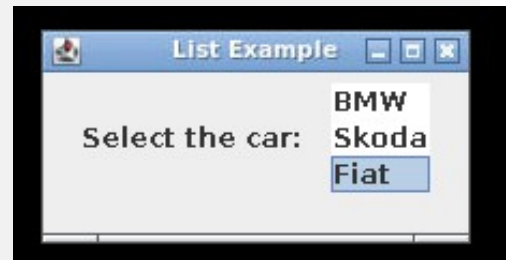
```

...
4  类 汽车 {
5
6      私人字符串 make;
7      public Car(String make) { this.make = make;
8          }@Override
9      public String toString() { return make; }
...
15 }
16 ...
17     Car cars[] = { new Car("BMW")、
18                     新汽车 ("斯柯达") 、
...                     new Car("Fiat") };

JList<Car> list = new JList<>(cars);

...

```



有问题或意见？

JC2002 Java 程序设计

第 9 天，第 2 课时：在模型中使用 JList 和 JTable

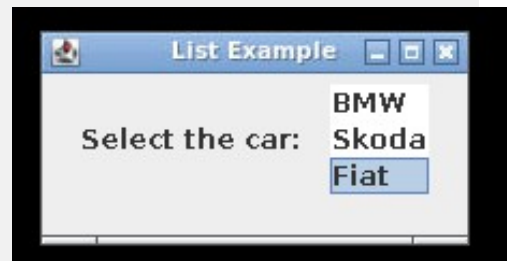
JList 的模型

- JList 有不同的预定义模型：
 - **ListModel**: 存储显示在列表中的数据和列表状态。要初始化 ListModel，您必须
 - 使用 **DefaultListModel** 类--一切都会为你搞定。
 - 扩展 **AbstractListModel** 类 - 管理数据并调用 "发射" 方法；必须实现从 ListModel 接口继承的 `getSize()` 和 `getElementAt()` 方法
 - 实施 ListModel 接口--由你管理一切
 - **ListSelectionModel**: 管理列表数据项的选择

使用 DefaultListModel 初始化 JList

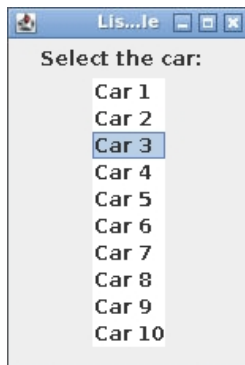
- 您可以使用 DefaultListModel 添加项目并初始化 JList

```
...  
4 类 汽车 {  
5  
6  私人字符串 make;  
7  public Car(String make) { this.make = make; }  
8  @Override  
9  public String toString() { return make; }  
... }  
15 ...  
16 DefaultListModel<Car> cars = new DefaultListModel<>();  
17 cars.addElement(new Car ("BMW")); cars.addElement(new  
18 Car ("Skoda")); cars.addElement(new Car ("Fiat"));  
19  
... JList<Car> list = new JList<>(cars);
```

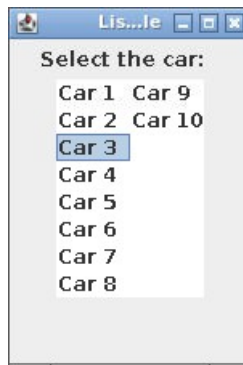


使用不同的 JList 布局

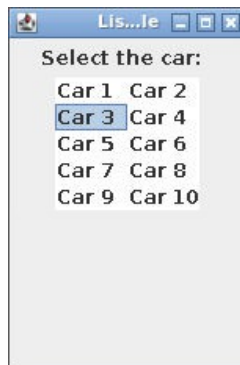
- 可以使用以下方法为 JList 选择不同的布局
设置布局方向()



垂直



垂直缠绕

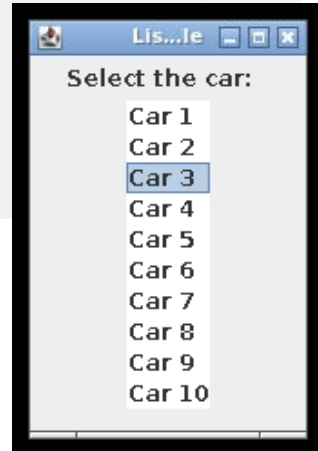


水平缠绕

垂直 JList 布局示例

- JList.VERTICAL 指定单列垂直布局（默认布局）

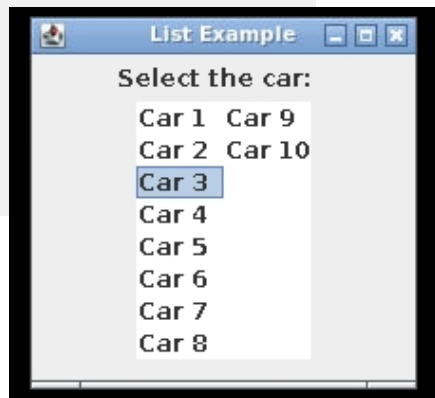
```
...    ...
15    DefaultListModel<Car> cars = new DefaultListModel<>();
16    for(int i=0; i<10; i++) {
17        cars.addElement(new Car("Car " + (i+1)));
18    }
19    JList<Car> list = new JList<>(cars);
20    ... list.setLayoutOrientation(JList.VERTICAL);
...    ...
```



垂直包络 JList 布局示例

- JList.VERTICAL_WRAP 表示 "报纸风格" 布局，其中包括细胞横向流动，然后纵向流动

```
...    ...  
15    DefaultListModel<Car> cars = new DefaultListModel<>();  
16    for(int i=0; i<10; i++) {  
17        cars.addElement(new Car("Car " + (i+1)));  
18    }  
19    JList<Car> list = new JList<>(cars);  
20    ... list.setLayoutOrientation(JList.VERTICAL_WRAP);  
...
```



垂直包络 JList 布局示例

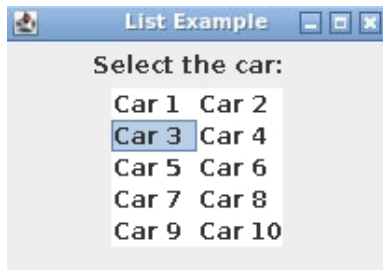
- JList.VERTICAL_WRAP 表示 "报纸风格" 布局，其中包括细胞横向流动，然后纵向流动

```
...  
15     DefaultListModel<Car> cars = new DefaultListModel<>();  
16     for(int i=0; i<10; i++) {  
17         cars.addElement(new Car("Car " + (i+1)));  
18     }  
19     JList<Car> list = new JList<>(cars);  
20     ... list.setLayoutOrientation(JList.HORIZONTAL_WRAP);  
...
```

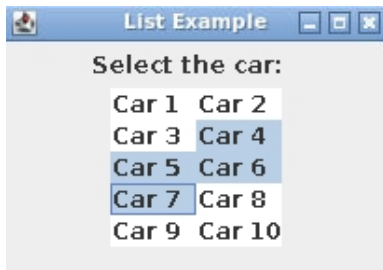


使用不同的 JList 选择模式

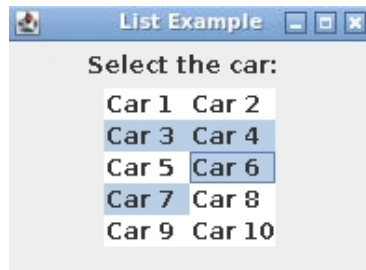
- 可以使用以下方法为 JList 选择不同的选择模式
设置选择模式



单选



单间隔选择

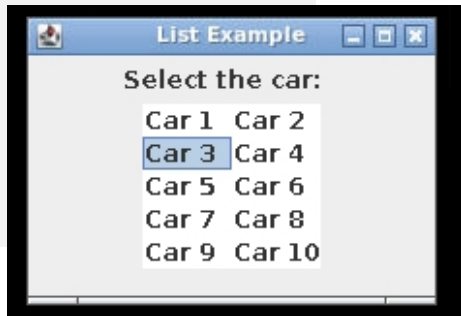


多间隔选择

单个选择示例

- ListSelectionModel.SINGLE_SELECTION 表示一次只能选择一个项目

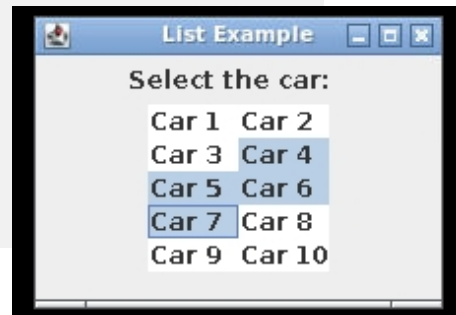
```
...  
15     DefaultListModel<Car> cars = new DefaultListModel<>();  
16     for(int i=0; i<10; i++) {  
17         cars.addElement(new Car("Car " + (i+1)));  
18     }  
19     JList<Car> list = new JList<>(cars);  
20     list.setLayoutOrientation(  
21         ListSelectionModel.SINGLE_SELECTION);  
...
```



单区间选择示例

- `ListSelectionModel.SINGLE_INTERVAL_SELECTION`
表示每次只能选择一个连续的区间

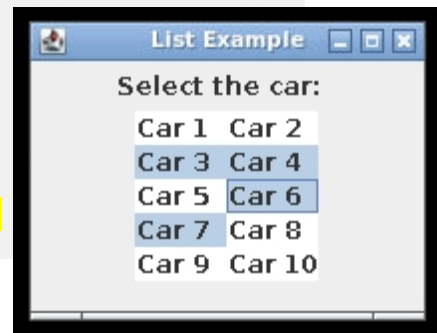
```
... ..
15     DefaultListModel<Car> cars = new DefaultListModel<>();
16     for(int i=0; i<10; i++) {
17         cars.addElement(new Car("Car " + (i+1)));
18     }
19     JList<Car> list = new JList<>(cars);
20     list.setLayoutOrientation(
21         ListSelectionModel.SINGLE_INTERVAL_SELECTION);
... ..
```



多区间选择示例

- ListSelectionModel.MULTIPLE_INTERVAL_SELECTION
表示可以自由选择项目（默认模式）

```
... ..
15     DefaultListModel<Car> cars = new DefaultListModel<>();
16     for(int i=0; i<10; i++) {
17         cars.addElement(new Car("Car " + (i+1)));
18     }
19     JList<Car> list = new JList<>(cars);
20     list.setLayoutOrientation(
21         ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
... ..
```



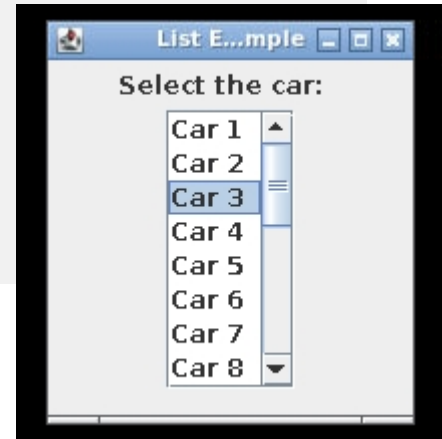
在列表中使用滚动条

- 一些元素（如 JLists）可以有很多项目，也可能有不适合预留的可见区域
 - 将该组件添加到滚动窗格中，即可启用滚动条
- **JScrollPane** 对象提供组件的可滚动视图
 - 您可以将 JList 对象（或任何其他 JComponent）添加到 JScrollPane 中对象作为参数传递给构造函数
 - 附加参数可用于进一步控制滚动条的行为（例如，滚动条是始终可见，还是仅在内容不适合可见区域时才可见）

带滚动条的列表示例 (1)

- 只需创建一个以列表为参数的滚动窗格，并将其添加到面板中即可

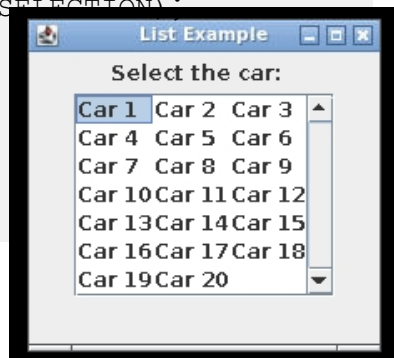
```
...    ...
15        DefaultListModel<Car> cars = new DefaultListModel<>();
16        for(int i=0; i<20; i++) {
17            cars.addElement(new Car("Car " + (i+1)));
18        }
19        JList<Car> list = new JList<>(cars);
20        JScrollPane listPane = new JScrollPane(list);
21
22        panel.add(label);
...    ... panel.add(listPane);
```



带滚动条的列表示例 (2)

- 使用另一个带有垂直和水平滚动条参数的构造函数，强制垂直滚动条始终可见

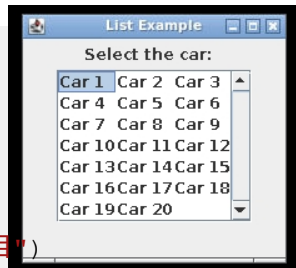
```
...  
15     JList<Car> list = new JList<>(cars);  
16  
17     list.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);  
18  
19     list.setLayoutOrientation(JList.HORIZONTAL_WRAP);  
20     JScrollPane listPane = new JScrollPane(list,  
21         ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,  
22         ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);  
...  
panel.add(label);  
panel.add(listPane);
```



添加列表选择监听器

- 要处理列表选择事件，请注册 `ListSelectionListener` 使用 `ListSelectionModel`
 - 检查事件的 `getValueIsAdjusting()` 以确保您的监听器不对错误类型的事件做出反应

```
...  
23 ListSelectionModel lsModel = list.getSelectionModel();  
24 lsModel.addListSelectionListener(new ListSelectionListener() {  
24     @Override  
26     public void valueChanged(ListSelectionEvent e) {  
27         if (e.getValueIsAdjusting() == false) {  
28             System.out.println("Item " + list.getSelectedIndex() + "项目")  
29                 "选定");  
30         }  
31     }  
32 });
```



```
$ java ListSelectionExample3
```

项目 2 已选定

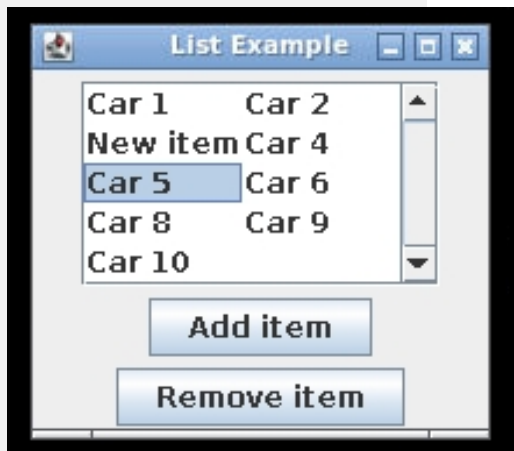
项目 16 已选定

添加和删除项目

- 可使用 ListModel 对象的 **remove(index)** 和 **insertElementAt(item,index)** 方法动态删除和添加列表项
 - 请注意，这些方法不会检查索引是否有效：您需要确保您没有从空列表中移除索引，或在列表末尾之外添加索引！

添加和删除项目示例

```
...
24     JButton removeButton = new JButton("Remove item");
25     JButton addButton = new JButton("Add item");
26     removeButton.addActionListener(new ActionListener() {
27         @Override
28         public void actionPerformed(ActionEvent e) {
29             cars.remove(list.getSelectedIndex());
30         }
31     });
32     addButton.addActionListener(new ActionListener() {
33         @Override
34         public void actionPerformed(ActionEvent e) {
35             cars.insertElementAt(new Car("New item"),
36                                 list.getSelectedIndex());
37         }
38     });
...
});
```



请注意，如果在未选择任何项目的情况下试图

删除该项目，则会出现异常！

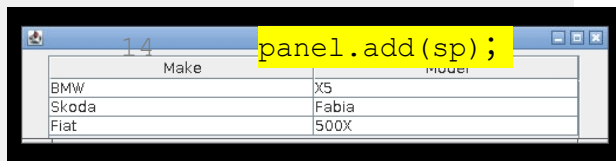
使用组件 JTable

- **JTable** 类允许您创建数据的表格视图
 - JTable 实例向用户展示一组排列整齐的项目以表格的形式列出行和列
 - 用户也可选择编辑表格数据
 - 表格可能会很复杂，我们将只讨论基本问题
- 可通过传递列名和数据直接初始化表格的 JTable 构造函数
 - 所有单元格均可编辑，数据将被视为字符串

- 这种方法只适用于事先已掌握数据的情况

直接使用 JTable 的简单示例

```
1  import java.awt.event.*;
2  import java.awt.*;
3  import javax.swing.*;
4  类 SimpleTableExample {
5      public static void main(String[] args) {
6          JFrame frame = new JFrame("Table Example");
7          JPanel panel = new JPanel();
8          String cols[] = {"品牌", "型号"};
9          String cars[][] = { {"BMW", "X5"},
10                               {"斯柯达", "法比亚"},
11                               {"菲亚特", "500X"} };
12          JTable table = new JTable(cars, cols);
13          JScrollPane sp = new JScrollPane(table);
```



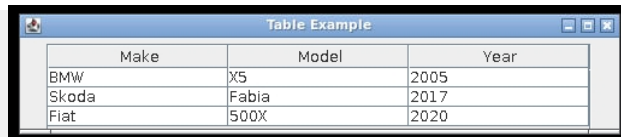
```
frame.add(panel); frame.setSize(500,100);  
frame.setVisible(true);  
}  
}
```


JTable 的模型

- 与 JList 一样：要使用表格模型，您必须
 - 使用 "**DefaultTableModel**" 类：一切都在您的掌控之中
 - 扩展 **AbstractTableModel** 类：管理数据并调用 "发射" 方法
 - 您必须执行 `getRowCount()`、`getColumnCount()` 和 `getValueAt()` 方法继承自 `TableModel` 接口。
 - 实施接口 **TableModel**：由你管理一切

通过自定义表格模型使用 JTable

```
1  import javax.swing.*;
2  导入 javax.swing.table.AbstractTableModel;
3  类 MyTableModel 扩展 AbstractTableModel {
4      private String[] columnNames = {"车型", "品牌", "年份"};
5      private Object[][] data = {"BMW", "X5", Integer.valueOf(2005)},
6                                  {"斯柯达", "法比亚", Integer.valueOf(2017)},
7                                  {"菲亚特", "500X", Integer.valueOf(2020)}};
8      公共 int getRowCount() {
9          return data.length;
10
11          18      }
12      20      }
13
14
15      16      public int getColumnCount() { return columnNames.length
17
18      ;
19
20      }
21      public String getColumnName(int col) { return
22
23      columnNames[col];
24      }
```



Make	Model	Year
BMW	X5	2005
Skoda	Fabia	2017
Fiat	500X	2020

```

public
Object 21 公共类 CustomTableModelExample {
getVal 22     public static void main(String[]
ueAt(i  args) {
nt      23         JFrame frame = new JFrame("Table
row,    Example");
int     24         JPanel panel = new JPanel();
col) {  25         MyTableModel model = new
return  MyTableModel();
data[r  26         JTable table = new JTable(model)
ow][co  ;
l];    27         JScrollPane sp = new
}      JScrollPane(table);

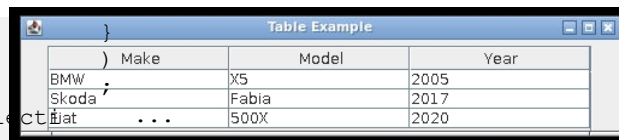
28         panel.add(sp);
29         frame.add(panel);
30         frame.setSize(500,100);
31         frame.setVisible(true);
32     }
33 }

```

使用带有选择监听器的表格

- 可以实现 `ListSelectionListener` 接口来监听选择事件，如用户选择单元格

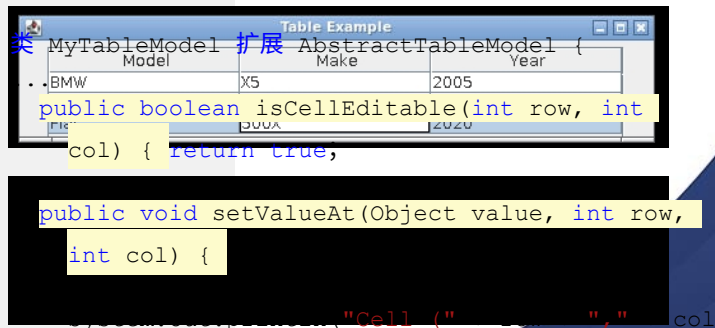
```
...
3
4  导入
5
6  javax.swing.event.ListSelectionEvent.ListSelectionEvent.ListSelectionEvent
7  onEvent.ListSelectionEvent.ListSelectionEvent
8
9  29  import
10
11  30  javax.swing.event.ListSelectionListener.ListSelectionListener
12
13  31
14  32
15  33  ListSelectionModel lsModel = table.getSelectionModel();
16  34  lsModel.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
17  35  lsModel.addListSelectionListener(new ListSelectionListener() {
18  36      public void valueChanged(ListSelectionEvent e) {
19  37          if (e.getValueIsAdjusting()) {
20  38              System.out.println("Selected: " + table.getValueAt(
21  39                  table.getSelectedRows()[0],
22  40                  table.getSelectedColumns()[0]).toString());
23  41
24  42  }
25  43  }
```



	Make	Model	Year
BMW	X5		2005
Skoda	Fabia		2017
Fiat	500X		2020

使用自定义模型的可编辑单元格

- 要使用从 AbstractTableModel 扩展的模型编辑单元格，需要实现 **isCellEditable()** 和 **setValueAt()** 方法



```
class MyTableModel extends AbstractTableModel {  
    Model  
    Make  
    Year  
    public boolean isCellEditable(int row, int  
    col) { return true,  
    public void setValueAt(Object value, int row,  
    int col) {  
        "Cell (" + row + ", " + col
```

```
+
"      ing());
单      }
元      ...
格
"
)
"
)

值
编
辑
:
"

+

v
a
l
u
e
.
t
o
S
t
r
```

```
$ java CustomTableModelExample2
```

单元格 (1,1) 已编辑值: 屋大薇

已选定: 500x



有问题或意见？

JC2002 Java 程序设计

第 9 天，第 3 节：并发基础知识

参考文献和学习目标

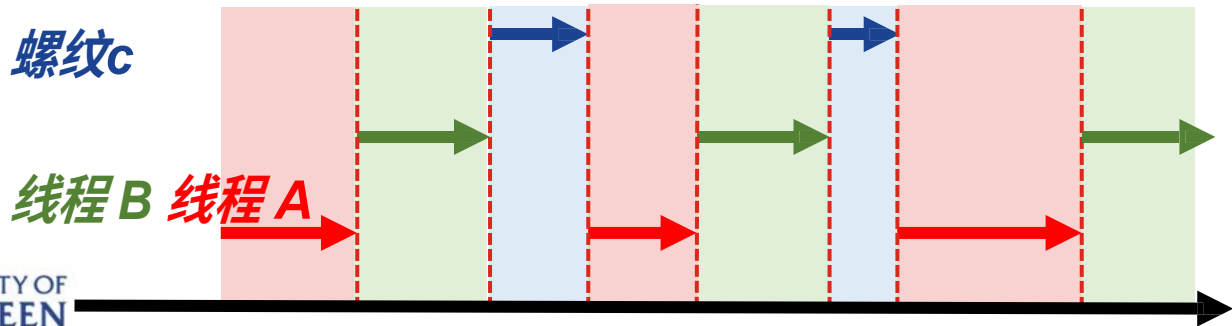
- 今天最后两节课的主要内容是：
 - Deitel, H., 《**Java 如何编程**》，**早期对象**，第 23 章，2018 年
- 经过今天最后两节课的学习，你们应该能够
 - 解释并发和多线程的概念
 - 使用 Thread 超类在 Java 中定义和使用线程
 - 使用 Swing API 在 Swing 应用程序中实现多线程功能

并行编程

- 在并发编程中，程序代码块（如方法）在重叠时段内*同时*执行
- 并发编程中有两个基本的执行单元：
 - **进程**：每个进程都有一个独立的执行环境（完整、私有的运行时资源集，即自己的内存空间）
 - **线程**：每个线程都存在于进程中（每个进程至少有一个线程），因此线程共享进程的资源，包括内存和打开的文件
 - 在 Java 编程中，我们主要关注线程

上下文切换

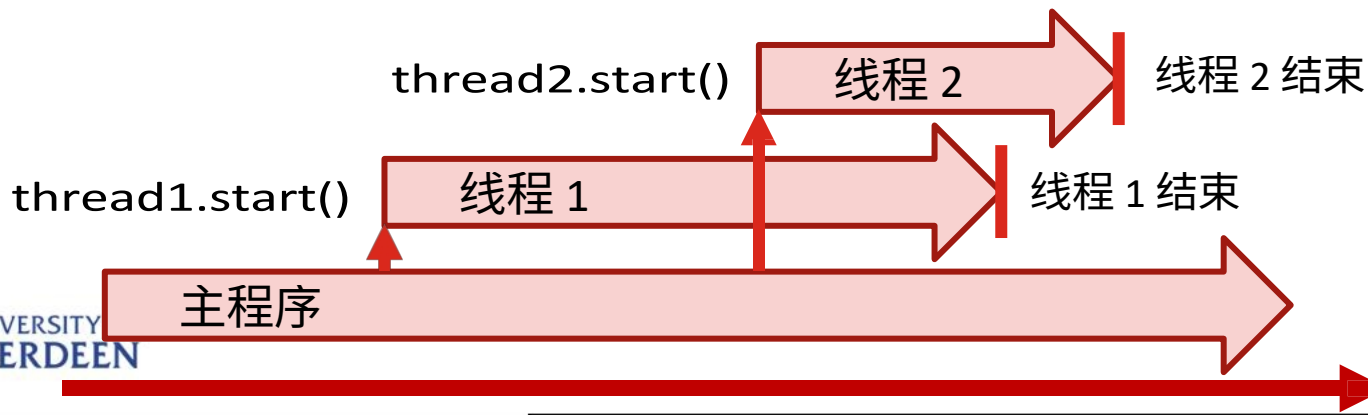
- 在操作系统中，多线程通常是通过以下方式实现的
 - 使用上下文切换
 - 线程以循环方式使用短时段运行（每个线程轮流运行），造成 CPU 多任务处理的假象



时间

Java 中的多线程

- 在 Java 中，可以通过扩展类 **Thread** 来使用线程
 - 要执行的代码在重载方法 **run()** 中实现
 - 使用方法 **start()** 启动线程



时间

简单的多线程示例 (1)

```
1 public class TestThreads {
2     static void printList(int n) {
3         for(int i=1; i<=5; i++) {
4             System.out.print(i*n + " ");
5         }
6         System.out.println();
7     }
8 }
9 public static void main(String args[]){
10     Thread thread1 = new Thread() {
11         public void run() {
12             TestThreads.printList(1);
```

```
12     }
13 };
14 线程 thread2 = new Thread() {
15     public void run() {
16         TestThreads.printList(10);
17     }
18 };
19 线程 1.start(); 线程 2.start();
20 }
21 }
22 }
```


简单的多线程示例 (2)

```
1  公共类 TestThreads {  
2      static void printList(int n) {  
3          for(int i=1; i<=5; i++) {  
4              System.out.print(i*n + " ");  
5          }  
6          System.out.println();  
7      }  
8      public static void main(String args[]){  
9          线程 thread1 = new Thread() {  
10             public void run() {  
11                 TestThreads.printList(1);
```

```
12     }  
13     };  
14     线程 thread2 = new Thread() {  
15         public void run() {  
16             TestThreads.printList(10);  
17         }  
18     };  
19     thread1.start();  
20     thread2.start();  
21 }  
22 }
```

类中的方法 **run()**

方法 **start()**

简单的多线程示例 (3)

```
1  公共类 TestThreads {
2      static void printList(int n) {
3          for(int i=1; i<=5; i++) {
4              System.out.print(i*n + " ");
5          }
6          System.out.println();
7      }
8      public static void main(String args[]) {
9          线程 thread1 = new Thread() {
10             public void
11             TestThreads.printList(1);
12         }
13     };
14     线程 thread2 = new Thread() {
15         public void run() {
16             TestThreads.printList(10);
17         }
18     };
19     线程 1.start(); 线程
20     2.start();
```

```
10 20 1 30 2
4 5
$
```

返回

两个线程都将运行各自的实例
方法 printList() 的并行
打印数字 4、
20、30、40

```
$ java TestThreads
```

1

、50

螺纹干扰 (1)

- 线程在访问同时使用相同的数据，导致内存不一致

```
类计数器 {  
    私人 int i = 0;  
    public void increment() {  
        i++;  
    }  
    ...  
}
```

调用增量()

主题 A

...

线程 B

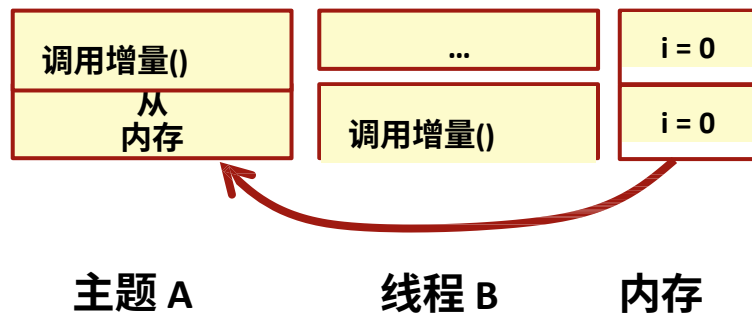
i = 0

内存

螺纹干扰 (2)

- 线程在访问同时使用相同的数据，导致内存不一致

```
类计数器 {  
    私人 int i = 0;  
    public void increment() {  
        i++;  
    }  
    ...  
}
```



螺纹干扰 (3)

- 线程在访问同时使用相同的数据，导致内存不一致

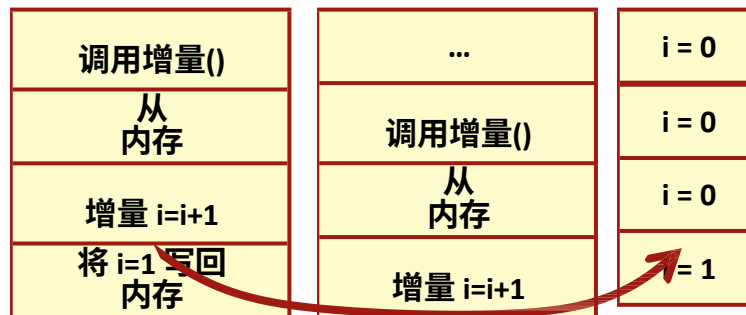
```
类计数器 {  
    私人 int i = 0;  
    public void increment() {  
        i++;  
    }  
    ...  
}
```



螺纹干扰 (4)

- 线程在访问同时使用相同的数据，导致内存不一致

```
类计数器 {  
    私人 int i = 0;  
    public void increment() {  
        i++;  
    }  
    ...  
}
```



主题 A

线程 B

内存

螺纹干扰 (5)

- 线程在访问
同时使用相同的数据，导致内存不一致



类计数器 {

 私人 int i = 0;

 public void increment() {

 i++;

 }

 ...

}

两个线程调用了 increment()、
但 i 只递增一次!



解决方案：同步

- 同步是线程干扰的一种解决方案

```
类 计数器 { 私有 int i
    = 0;
    public synchronized void increment() {
        i++;
    }
    ...
}
```

主题 A

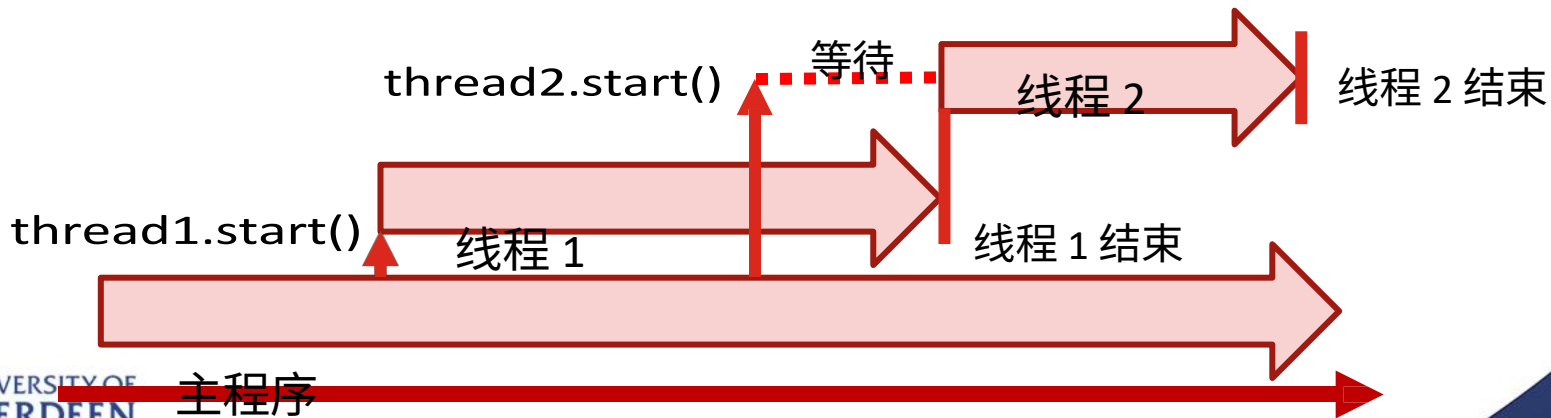
调用增量()
从 内存
增量 i=i+1
将 i=1 写回 内存
...

螺纹 B

...
调用增量()
受阻
受阻
从 内存

同步线程

- 当多个线程独立运行时，可能会出现以下情况迭出
 - 使用关键字 **synchronized** 会 "锁定" 方法的执行并强制其他线程等待直到执行完成



时间

同步多线程

```
1  公共类 TestThreads2 {  
2      synchronized static void printList(int n) {  
3          for(int i=1; i<=5; i++) {  
6              System.out.println();  
7          }  
8      }  
9      public static void main(String args[]){  
10         Thread thread1 = new Thread() {  
11             public void run() {  
                TestThreads2.printList(1);  
            }  
        }  
    }
```

```
12     }  
13     };  
14     线程 thread2 = new Thread() {  
15         }  
16     };  
17     };  
18     };  
19     };  
20     线程 1.start(); 线程 2.start();  
21     }  
22 }
```

等待线程 1
未始

```
$ java TestThreads2  
1 2 3 4 5  
10 20 30 40 50  
$
```

除了同步关键字外，该
示例与前一个示例相同！

使用 sleep() 的线程

```
1 public class TestThreads3 {  
2     static void countDown() {  
3         System.out.print("Seconds to launch: ");  
4         for(int i=10; i>0; i--) {  
5             System.out.print(i + " ");  
6             try {  
7                 Thread.sleep(1000);  
8             } catch (Exception e) {}  
9         }  
10        System.out.println("WHOOOSSH!");  
11    }  
12    public static void main(String args[]) {  
13        Thread thread1 = new Thread() {  
14            public void run() {  
15                TestThreads3.countDown();  
16            }  
17        };  
18    };  
19    thread1.start();  
20 }
```

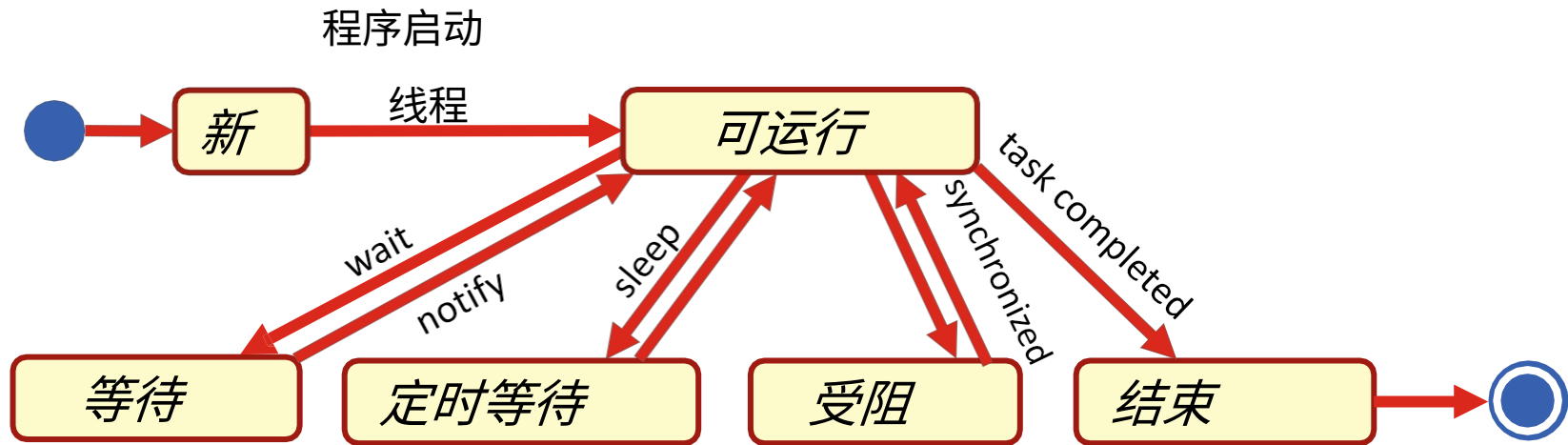
线程类的静态方法 sleep() 会暂时停止线程，并在参数设置的时间（毫秒）过后继续运行

数字将以一秒为间隔显示！

```
$ java TestThreads3  
发射前几秒: 10 9 8 7 6 5 4 3 2 1 WHOOOSSH!  
$
```

线程生命周期

- 线程可以处于不同的状态；终止后，线程不能重新开始（但是，您可以创建一个新的主题）



多线程中的中断

- 当 Java 线程处于等待状态时，例如在调用 `sleep()` 方法后，另一个线程可以尝试通过调用 **`interrupt()`** 方法来中断它：在这种情况下，会抛出 ***InterruptedException*** 异常。
 - *InterruptedException* 是一种经过检查的异常，因此在调用 `sleep()` 时需要使用异常处理程序（`try...catch` 结构）。

中断异常示例 (1)

```
1  公共类 TestThreads4 {
2      static void countDown(){
3          System.out.print("Seconds to launch: ");
4          for(int i=10; i>0; i--) {
5              System.out.print(i + " ");
6              try {
7                  Thread.sleep(1000);
8              } catch (InterruptedException e) {
9                  System.out.print("interrupt ");
10             }
11             System.out.println("WHOOOSSH!");
12         }
13     }
14     public static void main(String args[]){
15         线程 thread1 = new Thread() {
16             public void run() {
17                 TestThreads4.countDown();
18             }
19         };
20     }
```

在本例中，线程在处理完
InterruptedException 之后继续正
常运行

```
20         thread1.start();
21         thread1.interrupt();
22     }
23 }
```

中断异常示例 (2)

```
1  公共类 TestThreads4 {
2      static void countDown(){
3          System.out.print("Second
4          ;
5          for(int i=10; i>0; i--) {
6              System.out.print(i + " ");
7              try {
8                  Thread.sleep(1000);
9              } catch (InterruptedException e) {
10                 System.out.print("interrupt ");
11                 返回;
12             }
13             System.out.println("WHOOOSSH!");
14         }
15     }
16     public static void main(String args[]){
17         线程 thread1 = new Thread() {
18             public void run() {
19                 TestThreads4.countDown();
20             }
21         }
22     }
23 }
24 }
```

\$ java TestThreads3

发射时间: 10 秒钟

\$

在本例中，当
捕获中断异常

有问题或意见？

JC2002 Java 程序设计

第 9 天，第 4 节：Swing 的并发性

Swing 中的并发性

- 编写精良的 Swing 程序可利用并发功能创建永不 "冻结 "的用户界面，即程序始终对用户交互做出响应
 - 请注意，Swing 类中的大多数方法都不是 "线程安全 "的：您需要确保在同一线程中处理对这些方法的所有调用，以避免出现内存一致性错误
- Swing 程序员需要处理三种不同的线程：
 - 初始线程，执行初始应用代码

- *事件调度线程*，用于执行处理事件的代码
- 执行耗时的后台任务的*工作线程*

初始线程

- 在 Swing 程序中，初始线程通常只是创建一个实现 **Runnable** 接口的对象，用于初始化图形用户界面，并安排该对象在事件派发线程上执行。
 - 通过调用 **invokeLater()** 或 **SwingUtilities** 包的 **invokeAndWait()** 方法
 - 这两种方法都只接受一个参数，即定义了新任务
 - 这两种方法的不同之处在于，**invokeLater()** 只是调度任务并返回，

而 `invokeAndWait()` 则是等待调度的任务完成后才返回

InvokeLater 示例 (1)

```
1 import java.awt.event.*;
2 import javax.swing.*;
3
4 公共类 InvokeLaterExample {
5     private static void createAndShowGUI()
6     {
7         System.out.println("Creating
8         GUI..."); try {
9             Thread.sleep(1000);
10        } catch (Exception e) {
11            e.printStackTrace();
12        }
13        System.out.println("GUI 已创建! ");
14    }
15    public static void main(String[] args) {
16        javax.swing.SwingUtilities.invokeLater(new Runnable()
17        {
18            public void run() {
19                createAndShowGUI();
20            }
21        });
22    }
23}
```

让我们假设创建图形用户界面只需一秒钟

在调度新 Runnable 执行后，InvokeLater() 立即返回

```
$ java InvokeLaterExample
invokeLater() 已完成!
创建图形用户界面
```

```
System.out.println ("invokeLater() 已完成! ");
```

InvokeLater 示例 (2)

```
1  import java.awt.event.*;
2  import javax.swing.*;
3  公共类 InvokeLaterExample {
4      私人静态 void createAndShowGUI() {
5          System.out.println("Creating GUI...");
6          try {
7              Thread.sleep(1000);
8          } catch (Exception e) {
9              e.printStackTrace();
10         }
11         System.out.println("GUI 已创建! ");
12     }
13     public static void main(String[] args) {
14         javax.swing.SwingUtilities.invokeLater(new Runnable() {
15             public void run() {
16                 createAndShowGUI();
17             }
18         });
19     }
20 }
21 }
```

一秒后 createAndShowGUI() 执行
完毕

```
$ java InvokeLaterExample  
invokeLater() 已完成! 创建图形用户  
界面...  
创建了图形用户界面!  
$
```



InvokeAndWait 示例

```
1  import java.awt.event.*;
2  import javax.swing.*;
3  公共类 InvokeAndWaitExample {
4      私人静态 void createAndShowGUI() {
5          System.out.println("Creating GUI...");
6          try {
7              Thread.sleep(1000);
8          } catch (Exception e) {
9              e.printStackTrace();
10         }
11         System.out.println("GUI 已创建! ");
12     }
13     public static void main(String[] args) {
14         try {
15             javax.swing.SwingUtilities.invokeLaterAndWait(new Runnable() {
16                 public void run() {
17                     createAndShowGUI();
18                 }
19             });
20         } catch (Exception e) {
```

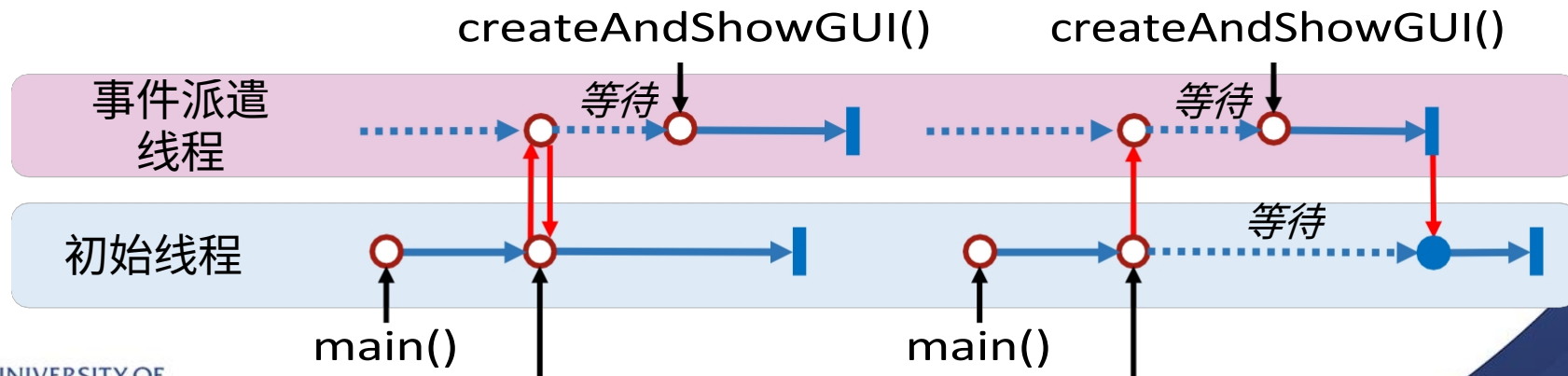
```
21             e.printStackTrace();
22         }
23         System.out.println("invokeAndWait()
24         completed!");
25     }
```

在 createAndShowGUI() 完成之前，
InvokeAndWait() 不会返回

```
$ java InvokeLaterExample 创建图形用户界面...  
创建了图形用户界面!  
invokeAndWait() 已完成!  
$
```


摘要： invokeLater 和 invokeAndWait

- 方法 `invokeLater()` 是异步的（即非阻塞），而 `invokeAndWait()` 是同步的（即阻塞）。



`invokeLater()`

`invokeAndWait()`

事件分派线程

- 由于大多数 Swing 对象方法都不是 "线程安全" 的，因此调用这些方法时会导致线程干扰的风险
 - 某些 Swing 组件方法在应用程序接口中标注为 "线程安全"。规范，并且可以从任何线程安全地调用它们。
 - 所有其他 Swing 组件方法都必须从 *事件派发线程* 调用
 - 忽略这一规则的程序可能在大多数情况下都能正常运行，但会出现难以跟踪和重现的不可预测的错误
 - 事件分派线程上的任务必须快速完成；否则，未处理的事件就会倒流，用户界面也会变得反应迟钝

事件分派线程示例

```
1 import java.awt.event.*;
2 import javax.swing.*;
3
4 公共类 EventDispatcherExample {
5     private static void createAndShowGUI() {
6         System.out.print("Creating GUI in " + Thread.currentThread());
7         System.out.println("Is event dispatch thread: " +
8             SwingUtilities.isEventDispatchThread());
9
10        JFrame frame = new JFrame("Event Dispatch
11        Demo"); frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLO
12        SE); JButton button = new JButton("Press Me!");
13        button.addActionListener(new ActionListener() {
14            public void actionPerformed(ActionEvent e) {
15                System.out.println("Button event in " + Thread.currentThread());
16                System.out.println("Is event dispatch thread: " +
17                    SwingUtilities.isEventDispatchThread());
18            }
19        });
20        ...
21    }
22 }
```

您可以使用静态方法

`currentThread()` 获取当前线程的信息

您可以使用静态方法

`isEventDispatchThread()` 来确定当前是否处于事件派发线程中



true

工作线程和 `SwingWorker`

- Swing 程序需要执行长期运行任务时，通常会使用 *工作线程*，也称为 *后台线程*
 - 在工作线程上运行的每个任务都由一个抽象类 `SwingWorker` 的子类
- `SwingWorker` 的生命周期涉及三个线程：
 - 当前线程（通常是事件派发线程）：调用 `执行()` 方法来调度 `SwingWorker` 的执行
 - 工作线程：调用 `doInBackground()` 方法，所有后台活动都应在此进行

- 事件调度线程：SwingWorker 调用 **process()** 和 **done()** 本主题的方法

SwingWorker 示例 (1)

```
1  import javax.swing.*;
2  import javax.swing.SwingUtilities.*;
3  import javax.swing.SwingWorker.*;
4  import java.awt.*;
5  import java.awt.event.*;
6  import java.beans.*;
7  公共类 SwingWorkerExample {
8      私有静态 SwingWorker createWorker() { 返回新
9          SwingWorker() {
10             @Override protected Boolean doInBackground() throws Exception {
11                 setProgress(0);
12                 for(int i=0; i<=100; i++) {
13                     Thread.sleep(500);
14                     setProgress(i);
15                 }
16             }
17             返回 false;
18         }
19     }
```

对于 SwingWorker 对象，您需要重载 doInBackground() 方法，以执行要执行的后台任务

在这个示例中，唯一的任务是增加一个计数器，并每秒更新进度两次

SwingWorker 示例 (2)

```
20  私人静态 void createAndShowGUI() {  
21      JFrame frame = new JFrame();  
22      JPanel panel = new JPanel();  
23      JButton button = new JButton("Start");  
24      JProgressBar progBar = new JProgressBar(0,100);  
25      progBar.setValue(0);  
26      progBar.setStringPainted(true);  
27      panel.add(button);  
28      panel.add(progBar);  
29      frame.add(panel);  
30      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
31      frame.setLocationRelativeTo(null);  
32      frame.setSize(250, 100);  
33      frame.setVisible(true);  
34      SwingWorker 工人;
```

创建带有按钮和进度条的用户界面

供以后使用的工人对象变量

SwingWorker 示例 (3)

35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54

为按钮添加
ActionListener，以便在按
下按钮时创建并执行
SwingWorker 对象

SwingWorker 示例 (4)

35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54

```
else if ("state".equals(e.getPropertyName())) {
```

为 Worker 添加
PropertyChangeListener，
以处理来自 Worker 线程的
进度和状态更新

当工作线程调用
setProgress() 方法时更
新进度条

工作线程完成后重新初始
化按钮

```
worker.execute();
```

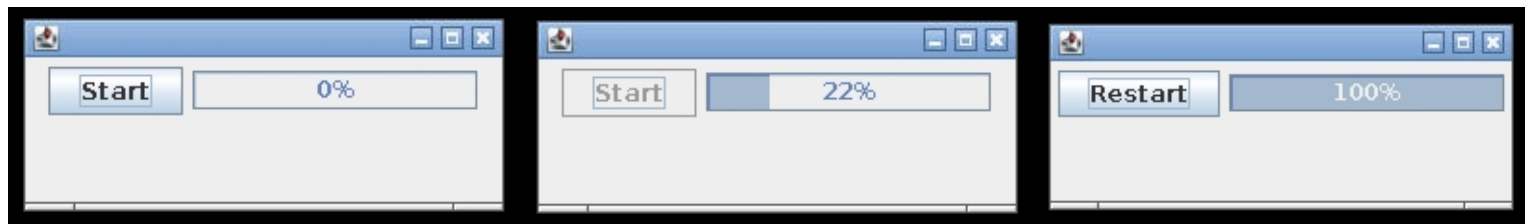
SwingWorker 示例 (5)

```
55     }  
56     });  
57 }  
58 public static void main(String[] args) {  
59     SwingUtilities.invokeLater(new Runnable() {  
60         public void run() {  
61             createAndShowGUI();  
62         }  
63     });  
64 }  
65 }
```

main() 方法调用

invokeLater() 创建图形用

户界面并启动应用程序



摘要

- 在 Swing 中，大多数 JComponent 类都预定义了以下模型存储与组件有关的数据
 - 模型有助于分离视图和相关数据
- 模型对于复杂的图形用户界面组件尤其有用，例如作为列表和表格
 - JList 和 JTable 组件有多种预定义模型
- 在并发编程中，代码块是同时执行的，通常使用多个线程

- 在 Java 中，线程是通过扩展类 Thread 或实现接口 Runnable

有问题或意见？