1495 UNIVERSITY OF ABERDEEN

JC2002 Java Programming

Day 9: Swing models, concurrency (CS)

Monday, 13 November

1495 UNIVERSITY OF ABERDEEN

JC2002 Java Programming

Day 9, Session 1: Models in Swing

References and learning objectives

- Today's first two sessions are mostly based on Oracle documentation:
  - https://docs.oracle.com/javase/tutorial/uiswing
- After the sessions, you should be able to:
  - Use Swing models in your Java GUI implementation
  - Implement custom functionalities in `JList` and `JTable` components

Swing models

- Models store the state of the component (e.g., mnemonics, whether it is enabled, selected, etc.) and data (e.g., items displayed in a list)
  - Most of the Swing components have predefined models
- Some components, such as lists, have multiple models
  - For example, `JList` uses `ListModel` and also `ListSelectionModel`
- For simple components (e.g., buttons) you would normally interact with the component directly, whereas for more complex components, such as lists and tables, interacting with models is a better choice

Why to use models?

- Models allow the separation of data from the view and controller if the MVC pattern is applied
- Default models can be extended and thus provide custom functionalities and flexibility in deciding how data is stored and retrieved
- Models automatically propagate changes to all registered listeners, allowing the view (i.e., GUI) to be updated

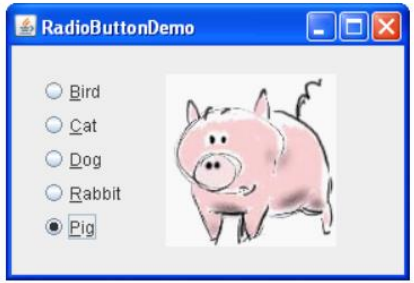
Using models vs. components directly

- There are different ways to achieve the same outcome in Java

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

// Use the component directly
System.out.println(pigButton.isSelected());

// Use the model
DefaultButtonModel model = (DefaultButtonModel)pigButton.getModel();
System.out.println(model.isSelected());
```

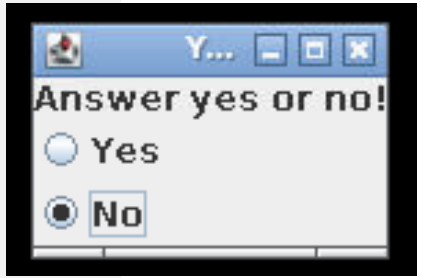


- Most component classes inherited from `JComponent` have a model by default, and it can be accessed using method `getModel()`

Interact with radio buttons directly

```java
import javax.swing.*;
public class YesNoButtonExample {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Yes or No?");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JPanel panel = new JPanel();
    BoxLayout boxlayout = new BoxLayout(panel, BoxLayout.Y_AXIS);
    panel.setLayout(boxlayout);
    JLabel question = new JLabel("Answer yes or no!");
    ButtonGroup group = new ButtonGroup();
    JRadioButton yes = new JRadioButton("Yes");
    JRadioButton no = new JRadioButton("No");
    group.add(yes); group.add(no);
    panel.add(question);
    panel.add(yes); panel.add(no);
    frame.add(panel);
    frame.pack();
    frame.setVisible(true);
```

```
$ java YesNoButtonExample
Yes selected: false
No selected: true
```

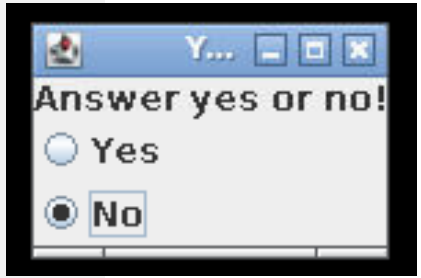



```java
    no.setSelected(true);
    System.out.println("Yes selected: "+yes.isSelected());
    System.out.println("No selected: "+no.isSelected());
  }
}
```

Interact with radio buttons via model

```java
import javax.swing.*;
public class YesNoButtonExample2 {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Yes or No?");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JPanel panel = new JPanel();
    BoxLayout boxlayout = new BoxLayout(panel, BoxLayout.Y_AXIS);
    panel.setLayout(boxlayout);
    JLabel question = new JLabel("Answer yes or no!");
    ButtonGroup group = new ButtonGroup();
    JRadioButton yes = new JRadioButton("Yes");
    JRadioButton no = new JRadioButton("No");
    DefaultButtonModel yesModel = (DefaultButtonModel)yes.getModel();
    DefaultButtonModel noModel = (DefaultButtonModel)no.getModel();
    group.add(yes); group.add(no);
    panel.add(question);
    panel.add(yes); panel.add(no);
    frame.add(panel);
    frame.pack();
    frame.setVisible(true);
```

```
$ java YesNoButtonExample
Yes selected: false
No selected: true
```




```java
    no.setSelected(true);
    System.out.println("Yes selected: "+yesModel.isSelected());
    System.out.println("No selected: "+noModel.isSelected());
  }
}
```

Defining custom button model

```java
import javax.swing.*;
class CustomButtonModel extends JToggleButton.ToggleButtonModel {
  private AbstractButton button;
  private String text;
  CustomButtonModel(AbstractButton button) {
    this.button = button;
    text = button.getText();
  }
  public void printStatus() {
    System.out.println(text + " selected: " + isSelected());
  }
  @Override
  public void setSelected(boolean b) {
    if(b) {
      button.setText(text + " (currently enabled)");
    }
    else {
      button.setText(text + " (currently disabled)");
    }
    super.setSelected(b);
  }
}
```

Custom radio button model should inherit toggle button model

New method for additional functionality

Overridend method for additional functionality

Using custom button model (1)

```
import javax.swing.*;
class CustomButtonModel extends JToggleButton.ToggleButtonModel {
  private AbstractButton button;
  private String text;
  Custo
    thi
    tex
  }
  publi
    Sys
  }
  @Over
  publi
    if(
      b
    }
    els
      b
    }
    sup
  }
}
```

```
public class YesNoButtonExample2 {
  public static void main(String[] args) {
...
    JRadioButton yes = new JRadioButton("Yes");
    JRadioButton 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();
  }
}
```

Instantiate custom models and assign to radio button objects

Using custom button model (2)

```
import javax.swing.*;
class CustomButtonModel extends JToggleButton.ToggleButtonModel {
  private AbstractButton button;
  private String text;
  Custo
    thi
    tex
  }
  publi
    Sys
  }
  @Over
  publi
    if(
      b
    }
    els
      b
    }
    sup
  }
}
```

```
public class YesNoButtonExample2 {
  public static void main(String[] args) {
...
    JRadioButton yes = new JRadioButton("Yes");
    JRadioButton 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();
  }
}
```

Use custom method `printStatus()`

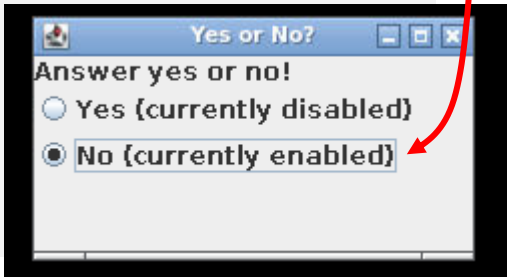
Using custom button model (3)

```
import javax.swing.*;
class CustomButtonModel extends JToggleButton.ToggleButtonModel {
  private AbstractButton button;
  private String text;
  Custo
    thi
    tex
  }
  publi
    Sys
  }
  @Over
  publi
    if(
      b
    }
    els
      b
    }
    sup
  }
}
```

```
23  public class YesNoButtonExample2 {
24    public static void main(String[] args) {
...   ...
32      JRadioButton yes = new JRadioButton("Yes");
33      JRadioButton no = new JRadioButton("No");
34      CustomButtonModel yesModel = new CustomButt
35      CustomButtonModel noModel = new CustomButto
36      yes.setModel(yesModel);
37      no.setModel(noModel);
...   ...
45      yes.setSelected(false);
46      no.setSelected(true);
47      yesModel.printStatus();
48      noModel.printStatus();
49    }
50  }
```

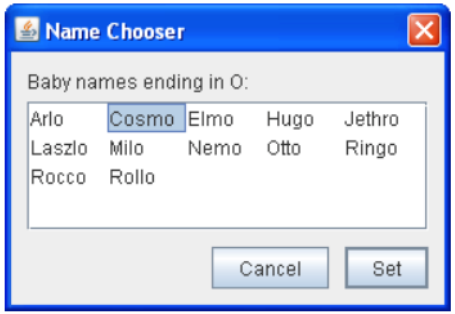
```
$ java YesNoButtonExample3
Yes selected: false
No selected: true
```

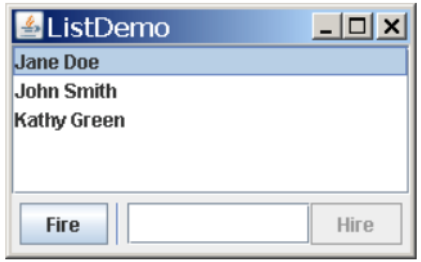
The text changes when the button is toggled

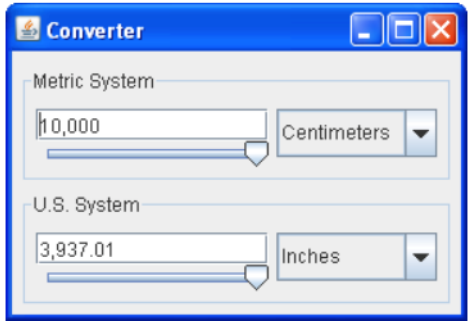


Using models for complex interaction

- The benefits of using models with simple components like `JButton` are usually limited, but with complex components, models are essential
  - With components such as **`JList`** and **`JTable`**, models allow more complex functionality and interaction
  - Models can also be beneficial for interaction between components



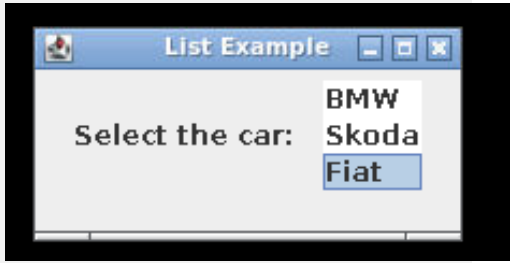




Simple example of using JList directly

- A `JList` instance presents the user with a group of items, displayed in one or more columns, to choose from

```
1   import java.awt.event.*;
2   import java.awt.*;
3   import javax.swing.*;
4   class SimpleListExample {
5     public static void main(String[] args) {
6       JFrame frame = new JFrame("List Example");
7       JPanel panel = new JPanel();
8       JLabel label = new JLabel("Select the car:  ");
9       String cars[]= {"BMW", "Skoda", "Fiat"};
10      Jlist<String> list = new Jlist<>(cars);
11      list.setSelectedIndex(2);
12      panel.add(label);
13      panel.add(list);
14      frame.add(panel);
15      frame.setSize(300,200);
16      frame.setVisible(true);
17    }
18  }
```

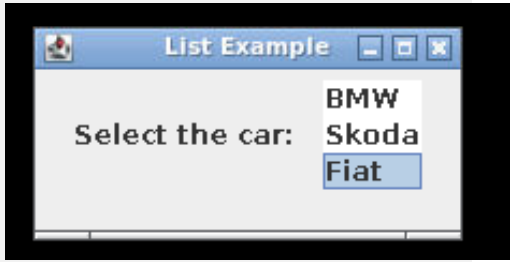


Note you need to define the type of items in `JList` (in this case, `String`)

Using JList directly with your own class

- You can store instances of your own class in `JList`, but you need to override `toString()` method to control how the items are displayed

```
...  ...
4    class Car {
5      private String make;
6      public Car(String make) { this.make = make; }
7      @Override
8      public String toString() { return make; }
9    }
...  ...
15       Car cars[]= { new Car("BMW"),
16                     new Car("Skoda"),
17                     new Car("Fiat") };
18       JList<Car> list = new JList<>(cars);
...  ...
```



Questions, comments?

1495 UNIVERSITY OF ABERDEEN

JC2002 Java Programming

Day 9, Session 2: Using JList and JTable with models

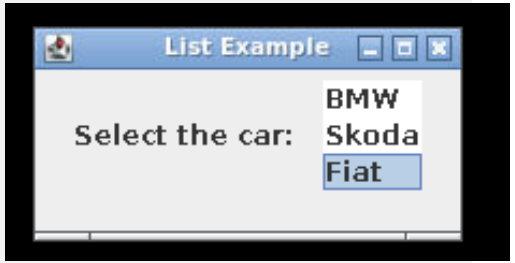
Models for JList

- There are different pre-defined models for `JList`:
  - **`ListModel`**: stores the information about the data items displayed in the list and the list states. To initialize a `ListModel`, you must either:
    - Use the class **`DefaultListModel`** — everything is taken care of for you.
    - Extend the class **`AbstractListModel`** — you manage the data and invoke the "fire" methods; you must implement `getSize()` and `getElementAt()` methods inherited from `ListModel` interface
    - Implement the `ListModel` interface — you manage everything
  - **`ListSelectionModel`**: manages the selection of list data items

Initialise JList using DefaultListModel

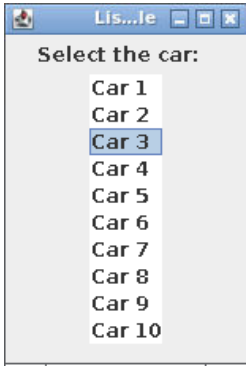
- You can use `DefaultListModel` to add items and initialise `JList`

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

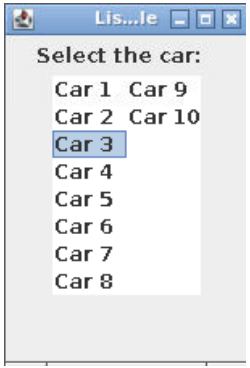


Using different JList layouts

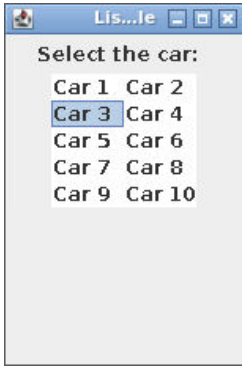
- Different layouts can be chosen for `JList` using method **`setLayoutOrientation()`**



VERTICAL



VERTICAL_WRAP

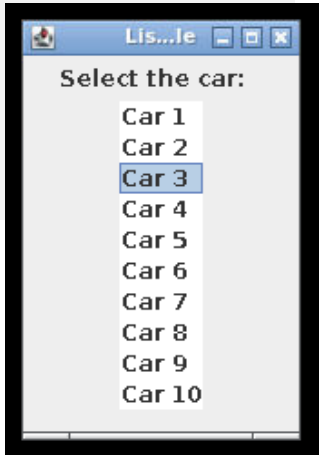


HORIZONTAL_WRAP

Vertical JList layout example

- `JList.VERTICAL` inidcates vertical layout in a single column (default layout)

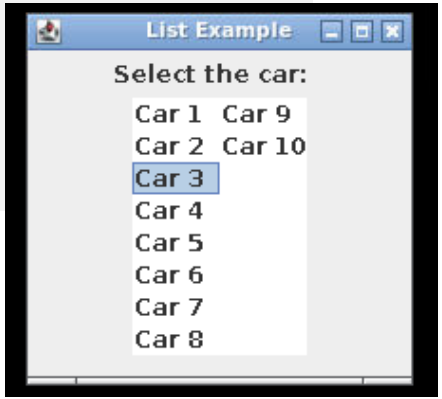
```
...
    DefaultListModel<Car> cars = new DefaultListModel<>();
    for(int i=0; i<10; i++) {
      cars.addElement(new Car("Car " + (i+1)));
    }
    JList<Car> list = new JList<>(cars);
    list.setLayoutOrientation(JList.VERTICAL);
...
```



Vertical wrap JList layout example

- `JList.VERTICAL_WRAP` indicates “newspaper style” layout with cells flowing horizontally, then vertically

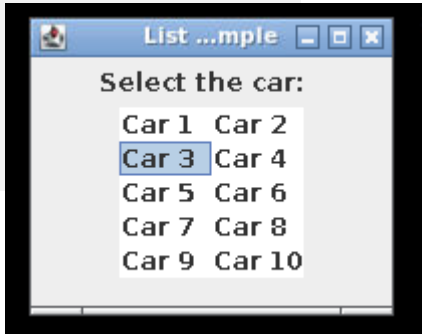
```
...
    DefaultListModel<Car> cars = new DefaultListModel<>();
    for(int i=0; i<10; i++) {
      cars.addElement(new Car("Car " + (i+1)));
    }
    JList<Car> list = new JList<>(cars);
    list.setLayoutOrientation(JList.VERTICAL_WRAP);
...
```



Vertical wrap JList layout example

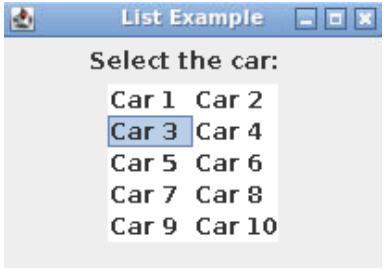
- `JList.VERTICAL_WRAP` indicates “newspaper style” layout with cells flowing horizontally, then vertically

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

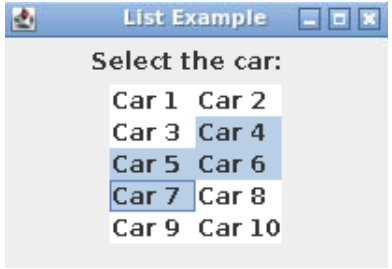


Using different JList selection modes

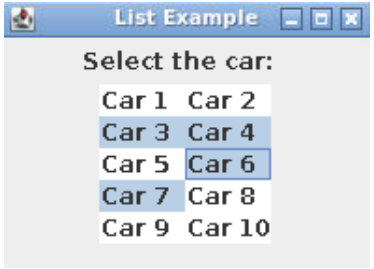
- Different selection modes can be chosen for `JList` using method **`setSelectionMode()`**



`SINGLE_SELECTION`



`SINGLE_INTERVAL_SELECTION`

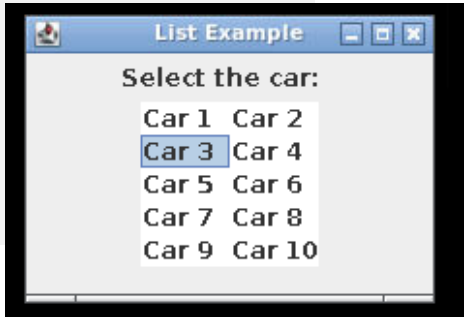


`MULTIPLE_INTERVAL_SELECTION`

Single selection example

- `ListSelectionModel.SINGLE_SELECTION` indicates that only one item can be selected at a time

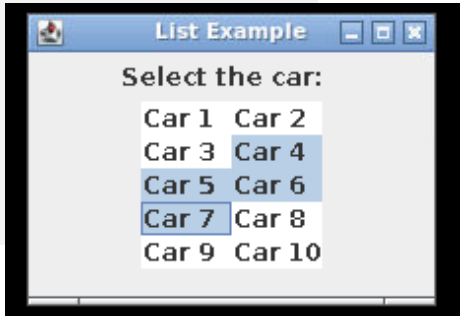
```
...
    DefaultListModel<Car> cars = new DefaultListModel<>();
    for(int i=0; i<10; i++) {
      cars.addElement(new Car("Car " + (i+1)));
    }
    JList<Car> list = new JList<>(cars);
    list.setLayoutOrientation(
      ListSelectionModel.SINGLE_SELECTION);
...
```



Single interval selection example

- `ListSelectionModel.SINGLE_INTERVAL_SELECTION` indicates that only one contiguous interval can be selected at a time

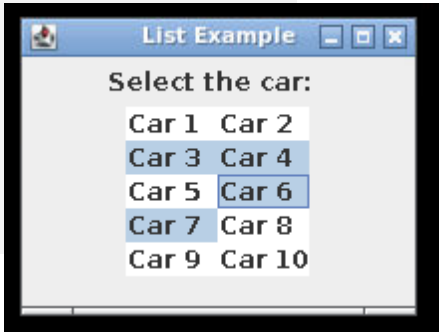
```
...
    DefaultListModel<Car> cars = new DefaultListModel<>();
    for(int i=0; i<10; i++) {
      cars.addElement(new Car("Car " + (i+1)));
    }
    JList<Car> list = new JList<>(cars);
    list.setLayoutOrientation(
      ListSelectionModel.SINGLE_INTERVAL_SELECTION);
...
```



Multiple interval selection example

- `ListSelectionModel.MULTIPLE_INTERVAL_SELECTION` indicates that items can be selected freely (default mode)

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



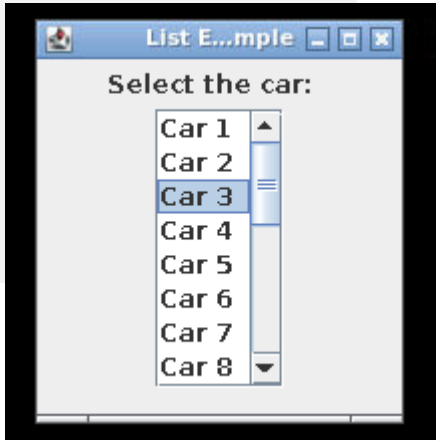
Using scroll bars with lists

- Some elements such as `JLists` can have many items and possible not fit in the visible area reserved for them
  - Scroll bars can be enabled by adding the component to a scroll pane
- A **`JScrollPane`** object provides a scrollable view of a component
  - You can add `JList` object (or any other `JComponent`) to `JScrollPane` object by passing it as a parameter to the constructor
  - Additional parameters can be used to further control the behavior of the scroll bar (e.g., whether the scroll bar is always visible, or only if the content does not fit in the visible area)

List with scroll bar example (1)

- Simply create a scroll pane with the list as a parameter, and add it to the panel

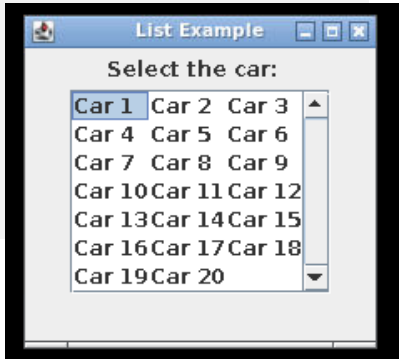
```
...
    DefaultListModel<Car> cars = new DefaultListModel<>();
    for(int i=0; i<20; i++) {
        cars.addElement(new Car("Car " + (i+1)));
    }
    JList<Car> list = new JList<>(cars);
    JScrollPane listPane = new JScrollPane(list);
    panel.add(label);
    panel.add(listPane);
...
```



List with scroll bar example (2)

- Force vertical scroll bar to be visible always using another constructor with parameters for vertical and horizontal scroll bars

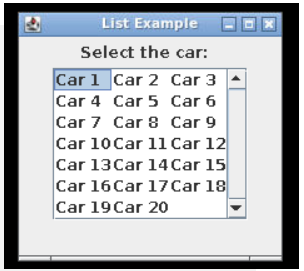
```
...
    JList<Car> list = new JList<>(cars);
    list.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
    list.setLayoutOrientation(JList.HORIZONTAL_WRAP);
    JScrollPane listPane = new JScrollPane(list,
        ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
        ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
    panel.add(label);
    panel.add(listPane);
...
```



Add a list selection listener

- To process list selection events, register `ListSelectionListener` using `ListSelectionModel`
  - Check the event's `getValueIsAdjusting()` to make sure your listener is not reacting on wrong type of events

```
...
    ListSelectionModel lsModel = list.getSelectionModel();
    lsModel.addListSelectionListener(new ListSelectionListener() {
      @Override
      public void valueChanged(ListSelectionEvent e) {
        if (e.getValueIsAdjusting() == false) {
          System.out.println("Item " + list.getSelectedIndex() +
                                 " selected");
        }
      }
    });
...
```



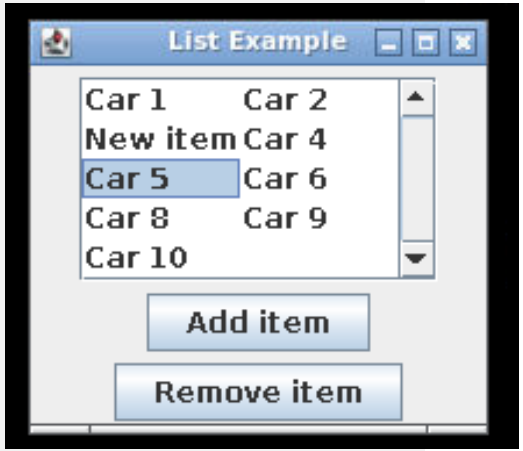

```
$ java ListSelectionExample3
Item 2 selected
Item 16 selected
Item 0 selected
```

Adding and removing items

- List items can be removed and added dynamically using **`remove(index)`** and **`insertElementAt(item,index)`** methods of `ListModel` object
  - Note that the methods do not check if the index is valid: you need to make sure you are not e.g., removing from an empty list, or adding beyond the end of the list!

Example of adding and removing items

```
...
    JButton removeButton = new JButton("Remove item");
    JButton addButton = new JButton("Add item");
    removeButton.addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent e) {
        cars.remove(list.getSelectedIndex());
      }
    });
    addButton.addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent e) {
        cars.insertElementAt(new Car("New item"),
                             list.getSelectedIndex());
      }
    });
...
```



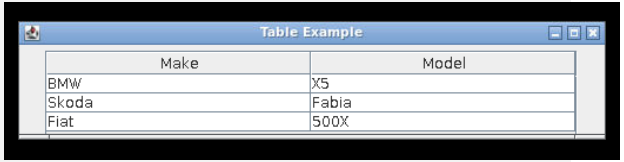
Note that exception is thrown, if you try to remove an item when none is selected!

Using component JTable

- **`JTable`** class allows you to create tabular views of your data
  - A `JTable` instance presents the user with a group of items arranged in a form of a table with rows and columns
  - User can be optionally also allowed to edit the table data
  - Tables can get complex, and we will just look at the basics
- A table can be initialised directly by passing the column names and data to the `JTable` constructor
  - All the cells will be editable, and data will be treated as `Strings`
  - This method is only suitable if you have the data available in advance

Simple example of using JTable directly

```
1   import java.awt.event.*;
2   import java.awt.*;
3   import javax.swing.*;
4   class SimpleTableExample {
5     public static void main(String[] args) {
6       JFrame frame = new JFrame("Table Example");
7       JPanel panel = new JPanel();
8       String cols[] = {“Make", "Model"};
9       String cars[][] = { {"BMW", "X5"},
10                          {"Skoda", "Fabia"},
11                          {"Fiat", "500X"} };
12      JTable table = new JTable(cars, cols);
13      JScrollPane sp = new JScrollPane(table);
14      panel.add(sp);
```



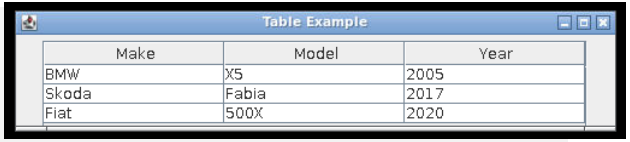

```
14      frame.add(panel);
15      frame.setSize(500,100);
16      frame.setVisible(true);
17    }
18  }
```

Models for JTable

- Same as with JList: to use a table model you must either:
  - Use the class **`DefaultTableModel`:** everything is taken care of for you
  - Extend the class **`AbstractTableModel`:** you manage the data and invoke the "fire" methods
    - You must implement `getRowCount()`, `getColumnCount()`, and `getValueAt()` methods inherited from the `TableModel` interface.
  - Implement the interface **`TableModel`**: you manage everything

Using JTable via custom table model

```java
import javax.swing.*;
import javax.swing.table.AbstractTableModel;
class MyTableModel extends AbstractTableModel {
  private String[] columnNames = {"Model", "Make", "Year"};
  private Object[][] data = {{"BMW", "X5", Integer.valueOf(2005)},
                             {"Skoda", "Fabia", Integer.valueOf(2017)},
                             {"Fiat", "500X", Integer.valueOf(2020)}};
  public int getRowCount() {
    return data.length;
  }
  public int getColumnCount() {
    return columnNames.length;
  }
  public String getColumnName(int col) {
    return columnNames[col];
  }
  public Object getValueAt(int row, int col) {
    return data[row][col];
  }
}
```

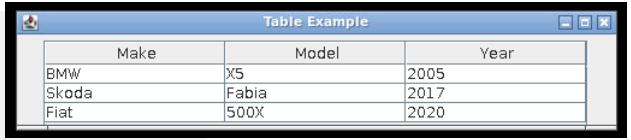



```java
public class CustomTableModelExample {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Table Example");
    JPanel panel = new JPanel();
    MyTableModel model = new MyTableModel();
    JTable table = new JTable(model);
    JScrollPane sp = new JScrollPane(table);
    panel.add(sp);
    frame.add(panel);
    frame.setSize(500,100);
    frame.setVisible(true);
  }
}
```

Using tables with selection listener

- Interface `ListSelectionListener` can be implemented to listen selection events, such as user selecting a cell

```
...  ...
3    import javax.swing.event.ListSelectionEvent;
4    import javax.swing.event.ListSelectionListener;
...  ...
29       ListSelectionModel lsModel = table.getSelectionModel();
30       lsModel.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
31       lsModel.addListSelectionListener(new ListSelectionListener() {
32         public void valueChanged(ListSelectionEvent e) {
33           if(e.getValueIsAdjusting()) {
34             System.out.println("Selected: " + table.getValueAt(
35                   table.getSelectedRows()[0],
36                   table.getSelectedColumns()[0]).toString());
37           }
38         }
39       });
...  ...
```

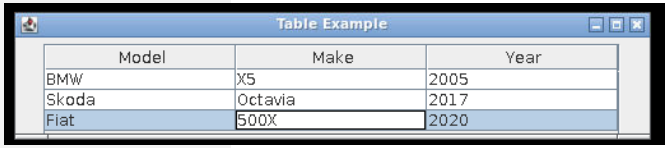



```
$ java CustomTableModelExample
Selected: BMW
Selected: Fabia
Selected: 500X
Selected: Skoda
Selected: BMW
Selected: 2017
Selected: Fiat
```

Using editable cells with a custom model

- To make cells editable with a model extended from `AbstractTableModel`, you need to implement **`isCellEditable()`** and **`setValueAt()`** methods

```
...
5   class MyTableModel extends AbstractTableModel {
...
22    public boolean isCellEditable(int row, int col) {
23      return true;
24    }
25    public void setValueAt(Object value, int row, int col) {
26      data[row][col] = value;
27      fireTableCellUpdated(row, col);
28      System.out.println("Cell (" + row + "," + col +
29        ") edited with value: " + value.toString());
30    }
...
```




```
$ java CustomTableModelExample2
Cell (1,1) edited with value: Octavia
Selected: 500X
```

Questions, comments?

1495 UNIVERSITY OF ABERDEEN

JC2002 Java Programming

Day 9, Session 3: Basics of concurrency

References and learning objectives

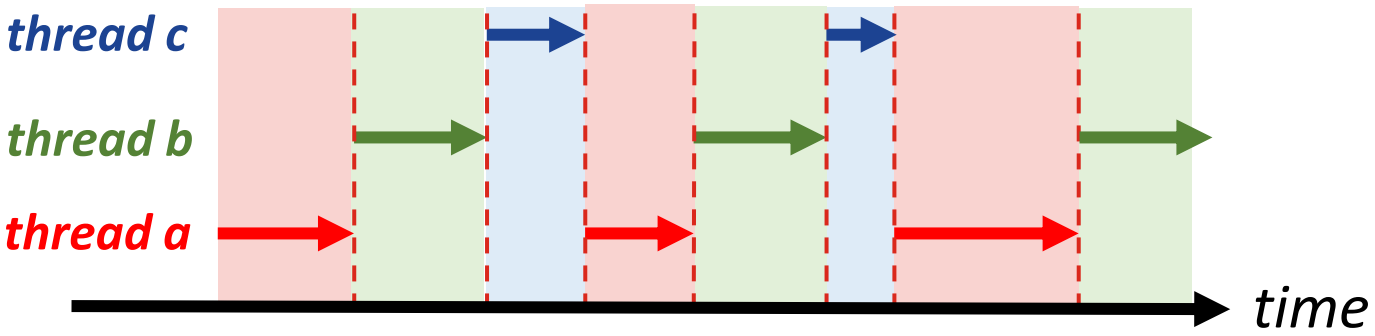
- Today's last two sessions are mostly based on:
  - Deitel, H., ***Java How to Program, Early Objects***, Chapter 23, 2018
- After today's last two sessions, you should be able to:
  - Explain the concepts of concurrency and multithreading
  - Define and use threads in Java using `Thread` superclass
  - Implement multithreading in Swing applications using Swing API

Concurrent programming

- In concurrent programming, blocks of program code (e.g., methods) are executed *concurrently* during overlapping time periods
- There are two basic units of execution in concurrent programming:
  - ***Processes***: each process has a self-contained execution environment (complete, private set of run-time resources, i.e., its own memory space)
  - ***Threads***: each thread exists within a process (every process has at least one thread) and therefore threads share the process's resources, including memory and open files
  - In Java programming, we are mostly concerned with threads

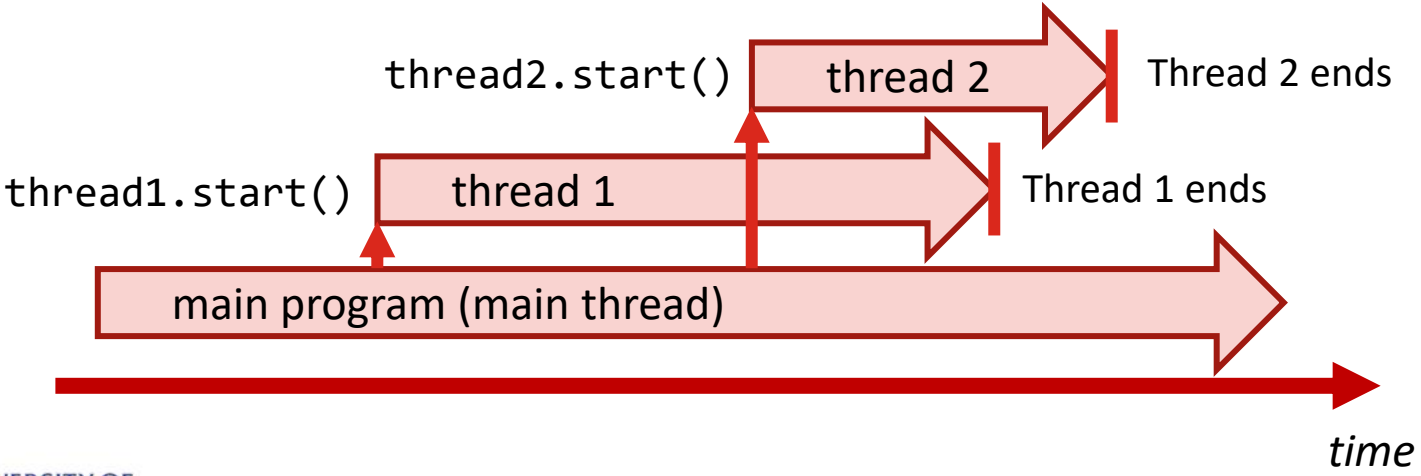
Context switching

- Typically, multithreading is implemented in operating systems by using *context switching*
  - Threads are run using short time slots in round robin fashion (each thread gets its turn alternatingly), creating illusion of CPU multitasking



Multithreading in Java

- In Java, threads can be used by extending class **`Thread`**
  - The code to be executed is implemented in overridden method **`run()`**
  - The thread is started using its method **`start()`**



Simple multithreading example (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     public static void main(String args[]){
9       Thread thread1 = new Thread() {
10        public void run() {
11          TestThreads.printList(1);
12      }
13        };
14        Thread thread2 = new Thread() {
15          public void run() {
16            TestThreads.printList(10);
17          }
18        };
19        thread1.start();
20        thread2.start();
21    }
22  }
```

Simple multithreading example (2)

```
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     public static void main(String args[]){
9       Thread thread1 = new Thread() {
10        public void run() {
11          TestThreads.printList(1);
12      }
13        };
14        Thread thread2 = new Thread() {
15        public void run() {
16          TestThreads.printList(10);
17        }
18        };
19        thread1.start();
20        thread2.start();
21    }
22  }
```

Implement threads by overriding method **`run()`** in class Thread

Start threads by using method **`start()`**

Simple multithreading example (3)

```java
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     public static void main(String args[]){
9       Thread thread1 = new Thread() {
10        public void run() {
11          TestThreads.printList(1);
12    }
13      };
14      Thread thread2 = new Thread() {
15        public void run() {
16          TestThreads.printList(10);
17        }
18      };
19      thread1.start();
20      thread2.start();
21    }
```

Prints numbers 1, 2, 3, 4, 5

Prints numbers 10, 20, 30, 40, 50

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

Both threads will run their own instance of method `printList()` in parallel

Thread interference (1)

- Threads may interfere with each other when they access the same data simultaneously, leading to memory inconsistency

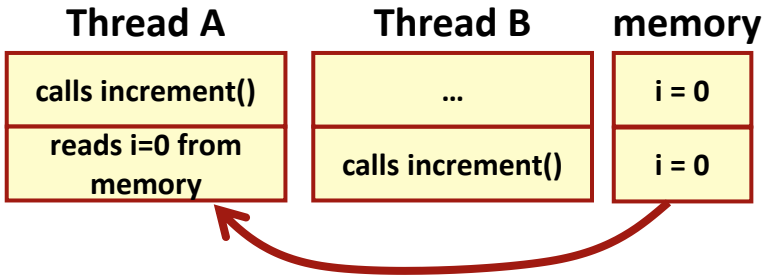
```
class Counter {
  private int i = 0;
  public void increment() {
    i++;
  }
  ...
}
```

Thread A	Thread B	memory
calls increment()	…	i = 0

Thread interference (2)

- Threads may interfere with each other when they access the same data simultaneously, leading to memory inconsistency

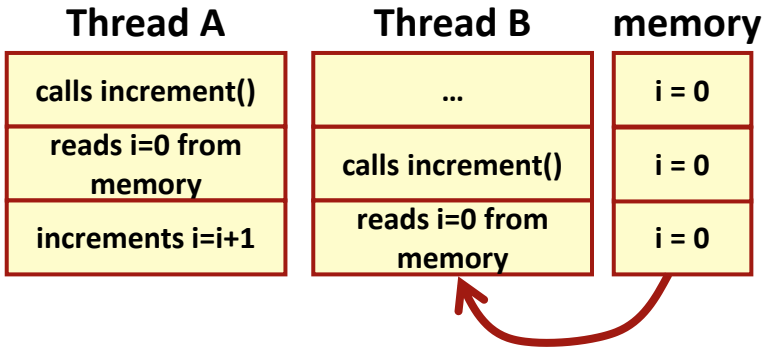
```
class Counter {
  private int i = 0;
  public void increment() {
    i++;
  }
  ...
}
```



Thread interference (3)

- Threads may interfere with each other when they access the same data simultaneously, leading to memory inconsistency

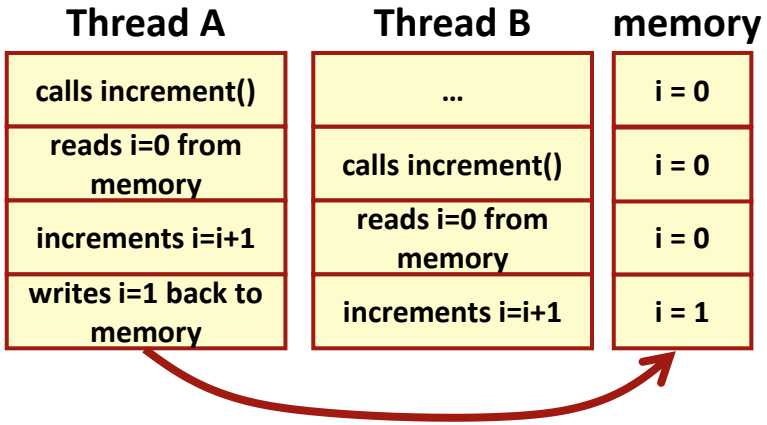
```
class Counter {
  private int i = 0;
  public void increment() {
    i++;
  }
  ...
}
```



Thread interference (4)

- Threads may interfere with each other when they access the same data simultaneously, leading to memory inconsistency

```
class Counter {
  private int i = 0;
  public void increment() {
    i++;
  }
  ...
}
```



Thread interference (5)

- Threads may interfere with each other when they access the same data simultaneously, leading to memory inconsistency

```
class Counter {
  private int i = 0;
  public void increment() {
    i++;
  }
  ...
}
```

Thread A	Thread B	memory
calls increment()	…	i = 0
reads i=0 from memory	calls increment()	i = 0
increments i=i+1	reads i=0 from memory	i = 0
writes i=1 back to memory	increments i=i+1	i = 1
…	writes i=1 back to memory	i = 1

Two threads invoked `increment()`, but `i` is incremented only once!

Solution: synchronisation

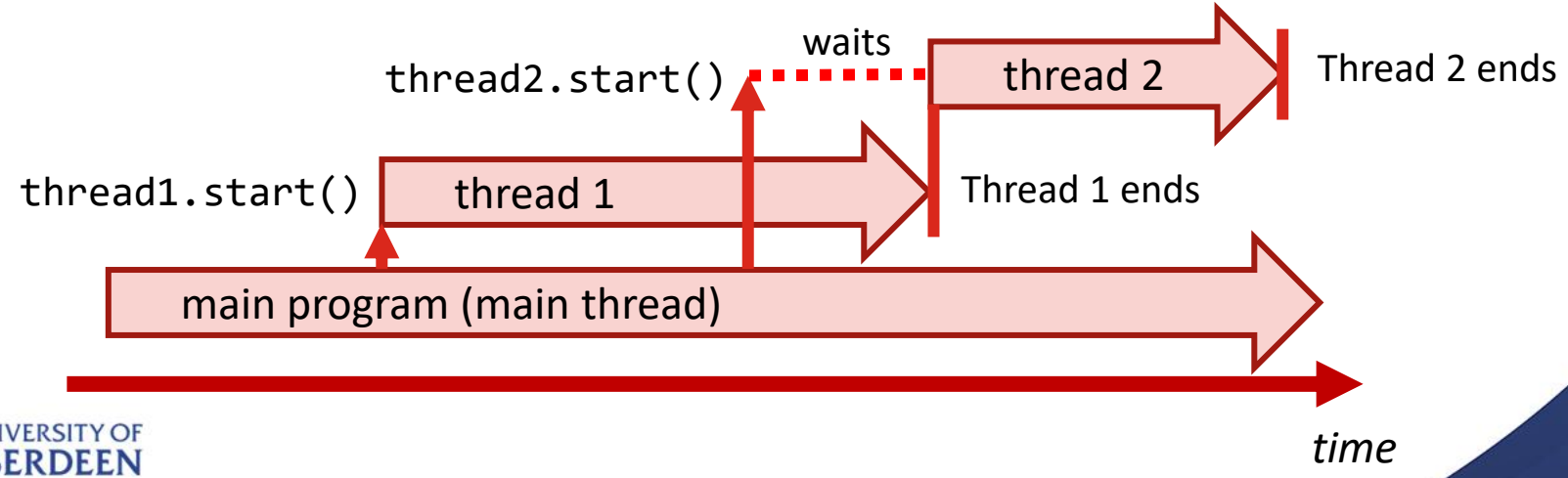
- Synchronisation is one solution to thread interference

```
class Counter {
  private int i = 0;
  public synchronized void increment() {
    i++;
  }
  ...
}
```

Thread A	Thread B
calls increment()	…
reads i=0 from memory	calls increment()
increments i=i+1	blocked
writes i=1 back to memory	blocked
…	reads i=0 from memory

Synchronising threads

- When multiple threads are running independently, things can happen in an unexpected order
  - Using keyword **`synchronized`** will "lock" the method execution and force other threads to wait until the execution completes



Multithreading with synchronized

```java
public class TestThreads2 {
  synchronized static void printList(int n) {
    for(int i=1; i<=5; i++) {
      System.out.print(i*n + " ");
    }
    System.out.println();
  }
  public static void main(String args[]){
    Thread thread1 = new Thread() {
      public void run() {
        TestThreads2.printList(1);
  }
    };
    Thread thread2 = new Thread() {
      public void run() {
        TestThreads2.printList(10);
      }
    };
    thread1.start();
    thread2.start();
  }
}
```

Waits for thread1 to finish before starting

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

Apart from keyword synchronized, this example is the same as the previous one!

Threads with sleep()

```java
public class TestThreads3 {
  static void countDown(){
    System.out.print("Seconds to launch: ");
    for(int i=10; i>0; i--) {
      System.out.print(i + " ");
      try {
        Thread.sleep(1000);
      } catch(Exception e) {}
    }
    System.out.println("WHOOOSSH!");
  }
  public static void main(String args[]){
    Thread thread1 = new Thread() {
      public void run() {
        TestThreads3.countDown();
      }
    };
    thread1.start();
  }
}
```

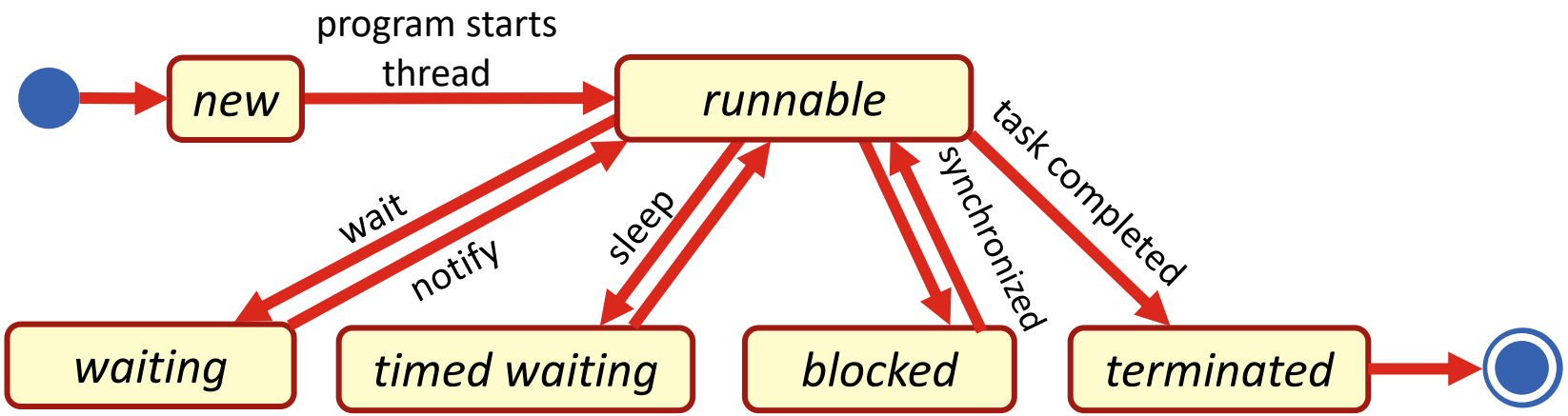
Static method `sleep()` of class Thread stops the thread temporarily and continues after the time given as parameter (in milliseconds) has passed

The numbers will appear with one second intervals!

```
$ java TestThreads3
Seconds to launch: 10 9 8 7 6 5 4 3 2 1 WHOOOSSH!
$
```

Thread life cycle

- Threads can be in different states; after termination, the thread cannot be started again (however, you can create a new thread)



Interruptions in multithreading

- When a Java thread is on a waiting state, e.g. after invoking `sleep()`, another thread can try to interrupt it by invoking its **`interrupt()`** method: in this case, ***`InterruptedException`*** is thrown
  - *`InterruptedException`* is a checked exception, so exception handler (`try…catch` structure) is required when `sleep()` is invoked

Example of InterruptedException (1)

```java
public class TestThreads4 {
  static void countDown(){
    System.out.print("Seconds to launch: ");
    for(int i=10; i>0; i--) {
      System.out.print(i + " ");
      try {
        Thread.sleep(1000);
      } catch(InterruptedException e) {
        System.out.print("interrupt ");
      }
      System.out.println("WHOOOSSH!");
    }
  }
  public static void main(String args[]){
    Thread thread1 = new Thread() {
      public void run() {
        TestThreads4.countDown();
      }
    };
    thread1.start();
    thread1.interrupt();
  }
}
```

In this example, the thread continues running normally after `InterruptedException` is handled

```
$ java TestThreads3
Seconds to launch: 10 interrupt 9 8 7 6 5 4 3 2 1 WHOOOSSH!
$
```

Example of InterruptedException (2)

```java
public class TestThreads4 {
  static void countDown(){
    System.out.print("Seconds to launch: ");
    for(int i=10; i>0; i--) {
      System.out.print(i + " ");
      try {
        Thread.sleep(1000);
      } catch(InterruptedException e) {
        System.out.print("interrupt ");
        return;
      }
      System.out.println("WHOOOSSH!");
    }
  }
  public static void main(String args[]){
    Thread thread1 = new Thread() {
      public void run() {
        TestThreads4.countDown();
      }
    };
    thread1.start();
    thread1.interrupt();
  }
}
```

In this example, the thread ends when `InterruptedException` is catched

```
$ java TestThreads3
Seconds to launch: 10 interrupt
$
```

Questions, comments?

1495 UNIVERSITY OF ABERDEEN

JC2002 Java Programming

Day 9, Session 4: Concurrency with Swing

Concurrency in Swing

- A well-written Swing program uses concurrency to create a user interface that never “freezes“, i.e., the program is always responsive to user interaction
  - Note that most of the methods in Swing classes are not “thread safe”: you need to ensure that all calls to them are handled in the same thread to avoid memory consistency errors
- Swing programmer deals with three different kinds of threads:
  - *Initial threads*, where the initial application code is executed
  - *Event dispatch threads*, where the code for handling events is executed
  - *Worker threads*, where time-consuming background tasks are executed

Initial threads

- In Swing programs, the initial threads typically just create an object implementing **`Runnable`** interface that initializes the GUI and schedule that object for execution on the event dispatch thread
  - GUI creation task is scheduled by invoking either **`invokeLater()`** or **`invokeAndWait()`** method of SwingUtilities package
  - Both methods take a single argument, i.e., `Runnable` object defining the new task
  - The methods differ in that `invokeLater()` simply schedules the task and returns, whereas `invokeAndWait()` waits for the scheduled task to finish before returning

InvokeLater example (1)

```java
import java.awt.event.*;
import  javax.swing.*;
public class InvokeLaterExample {
  private static void createAndShowGUI() {
    System.out.println("Creating GUI...");
    try {
      Thread.sleep(1000);
    } catch(Exception e) {
      e.printStackTrace();
    }
    System.out.println("GUI created!");
  }
  public static void main(String[] args) {
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        createAndShowGUI();
      }
    });
    System.out.println("invokeLater() completed!");
  }
}
```

Let us play that it takes one second to create GUI

`InvokeLater()` returns immediately after scheduling the new `Runnable` to execute

```
$ java InvokeLaterExample
invokeLater() completed!
Creating GUI...
```

InvokeLater example (2)

```java
import java.awt.event.*;
import  javax.swing.*;
public class InvokeLaterExample {
  private static void createAndShowGUI() {
    System.out.println("Creating GUI...");
    try {
      Thread.sleep(1000);
    } catch(Exception e) {
      e.printStackTrace();
    }
    System.out.println("GUI created!");
  }
  public static void main(String[] args) {
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        createAndShowGUI();
      }
    });
    System.out.println("invokeLater() completed!");
  }
}
```

`createAndShowGUI()` completes execution one second later

```
$ java InvokeLaterExample
invokeLater() completed!
Creating GUI...
GUI created!
$
```

InvokeAndWait example

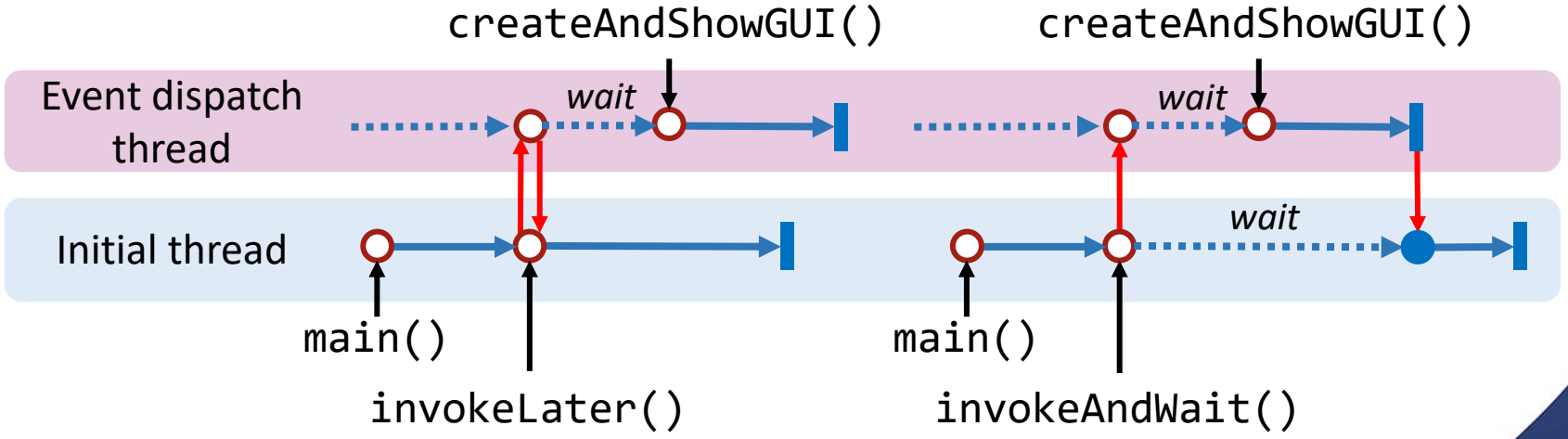
```java
import java.awt.event.*;
import javax.swing.*;
public class InvokeAndWaitExample {
  private static void createAndShowGUI() {
    System.out.println("Creating GUI...");
    try {
      Thread.sleep(1000);
    } catch(Exception e) {
      e.printStackTrace();
    }
    System.out.println("GUI created!");
  }
  public static void main(String[] args) {
    try {
      javax.swing.SwingUtilities.invokeAndWait(new Runnable() {
        public void run() {
          createAndShowGUI();
        }
      });
    } catch(Exception e) {
      e.printStackTrace();
    }
    System.out.println("invokeAndWait() completed!");
  }
}
```

`InvokeAndWait()` does not return before `createAndShowGUI()` has completed

```
$ java InvokeLaterExample
Creating GUI...
GUI created!
invokeAndWait() completed!
$
```

Summary: invokeLater and invokeAndWait

- Method `invokeLater()` is *asynchronous* (i.e., *non-blocking*), whereas `invokeAndWait()` is *synchronous* (i.e., *blocking*)



Event dispatch threads

- Since most Swing object methods are not “thread safe”, invoking them from multiple threads causes a risk of thread interference
  - Some Swing component methods are labelled “thread safe” in the API specification, and they can be safely invoked from any thread
  - All the other Swing component methods must be invoked from the *event dispatch thread*
    - Programs ignoring this rule may function correctly most of the time, but are prone to unpredictable errors that are difficult to track and reproduce
    - Tasks on the event dispatch thread must finish quickly; if they do not, unhandled events back up and the user interface becomes unresponsive

Event dispatch thread example

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

You can use static method `currentThread()` to get information about the current thread

You can use static method `isEventDispatchThread()` to determine whether you are currently in the event dispatch thread


```
$ java EventDispatcherExample
Creating GUI in Thread[AWT-EventQueue-0,6,main]
Is event dispatch thread: true
Button event in Thread[AWT-EventQueue-0,6,main]
Is event dispatch thread: true
$
```

Worker threads and SwingWorker

- When a Swing program needs to execute a long-running task, it usually uses a *worker thread*, also known as *background thread*
  - Each task running on a worker thread is represented by an instance of a subclass of abstract class **`SwingWorker`**
- Three threads are involved in the life cycle of a `SwingWorker`:
  - Current thread (often event dispatch thread): calls the **`execute()`** method to schedule execution of `SwingWorker`
  - Worker thread: calls **`doInBackground()`** method, where all the background activity should happen
  - Event dispatch thread: `SwingWorker` invokes **`process()`** and **`done()`** methods on this thread

SwingWorker example (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   public class SwingWorkerExample {
8     private static SwingWorker createWorker() {
9       return new 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          return false;
17        }
18      };
19    }
```

For your `SwingWorker` object, you need to override `doInBackground()` method to implement the background task to be executed

In this example, the only task is to increase a counter and update progress two times per second

SwingWorker example (2)

```
20    private static 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 worker;
```

Create UI with a button and progress bar

Worker object variable for later use

SwingWorker example (3)

```
35      button.addActionListener(new ActionListener() {
36        @Override public void actionPerformed(ActionEvent e) {
37          button.setEnabled(false);
38          progBar.setValue(0);
39          SwingWorker worker = createWorker();
40          worker.addPropertyChangeListener(
41            new PropertyChangeListener() {
42              public void propertyChange(PropertyChangeEvent e) {
43                if ("progress".equals(e.getPropertyName())) {
44                  progBar.setValue((Integer)e.getNewValue());
45                }
46                else if ("state".equals(e.getPropertyName())) {
47                  if (e.getNewValue() == StateValue.DONE) {
48                    button.setText("Restart");
49                    button.setEnabled(true);
50                  }
51                }
52              }
53          });
54          worker.execute();
```

Add `ActionListener` to the button to create and execute `SwingWorker` object when the button is pressed

SwingWorker example (4)

```
35      button.addActionListener(new ActionListener() {
36        @Override public void actionPerformed(ActionEvent e) {
37          button.setEnabled(false);
38          progBar.setValue(0);
39          SwingWorker worker = createWorker();
40          worker.addPropertyChangeListener(
41            new PropertyChangeListener() {
42              public void propertyChange(PropertyChangeEvent e) {
43                if ("progress".equals(e.getPropertyName())) {
44                  progBar.setValue((Integer)e.getNewValue());
45                }
46                else if ("state".equals(e.getPropertyName())) {
47                  if (e.getNewValue() == StateValue.DONE) {
48                    button.setText("Restart");
49                    button.setEnabled(true);
50                  }
51                }
52              }
53          });
54          worker.execute();
```

Add `PropertyChangeListener` to the worker to handle progress and status updates from the worker thread

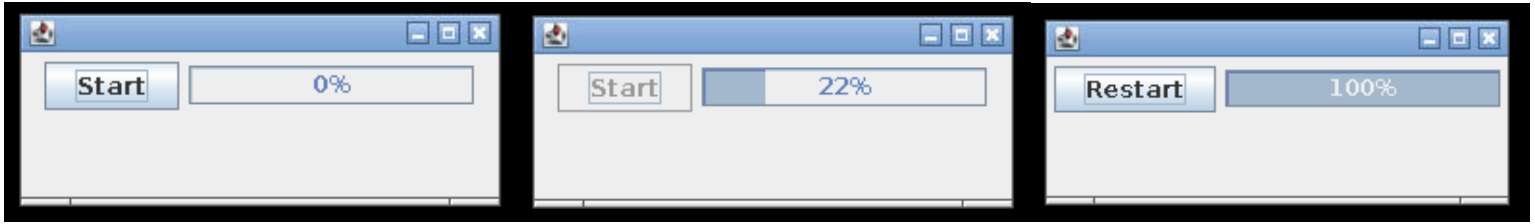
Update progress bar when the worker thread calls `setProgress()` method

Reinitialise button when the worker thread is done

SwingWorker example (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    }
```

The `main()` method calls `invokeLater()` to create GUI and start the application



Summary

- In Swing, most `JComponent` classes have pre-defined models for storing data related to components
  - Models help to separate the view and the related data
- Models are especially useful for complex GUI components, such as lists and tables
  - `JList` and `JTable` components have multiple predefined models
- In concurrent programming, blocks of code are executed simultaneously, typically by using multiple threads
  - In Java, threads are defined and used by extending class `Thread` or implementing interface Runnable

Questions, comments?