ENTERPRISE JAVA PROGRAMMING / PROGRAMMING II

MODULES CT545 / CT875, SEMESTER 2, ACADEMIC YEAR 2020-2021
NATIONAL UNIVERSITY OF IRELAND, GALWAY

Lecture 5b

Lecturer: Dr Patrick Mannion

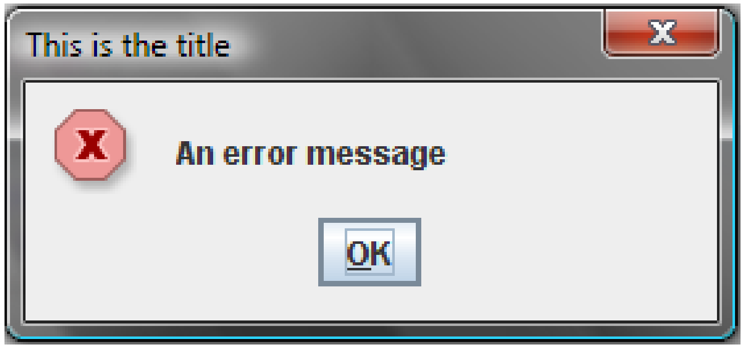
TOPICS

- Graphical User Interfaces (part 2)
  - *Some more Swing*
  - *Java2D* (very briefly)
  - *Model-View-Controller* pattern
- *Nested classes*

USER INTERFACES
Swing

- Just briefly, a few useful Swing functions…
- For simple modal dialogue windows, you don't need to program components and event handling yourself. Example:

```
 JOptionPane.showMessageDialog(parentFrame,
     "An error message",
     "This is the title",
     JOptionPane.ERROR_MESSAGE);  // Icon type. There are warning,
   plain and information icons too
```

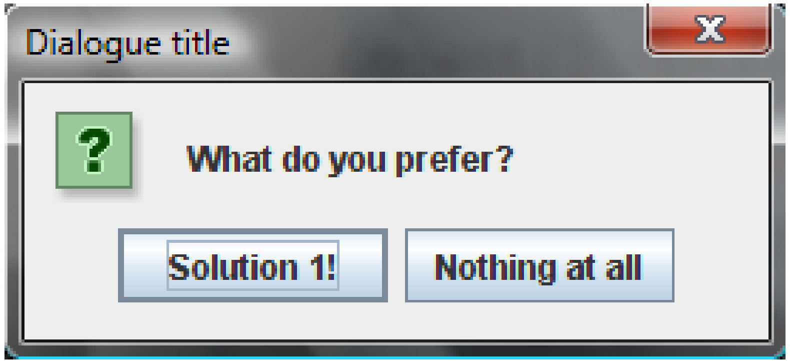

This is the title

An error message

USER INTERFACES
Swing

- Creating simple choice dialogs is very easy:

```
Object[] options = {"Solution 1!", "Nothing at all"};
int n = JOptionPane.showOptionDialog(parentFrame,  // n = result
    "What do you prefer?", "Dialogue title",
    JOptionPane.YES_NO_OPTION,
    JOptionPane.QUESTION_MESSAGE,
    null, options,  //button labels
    options[0]); //default button
```



USER INTERFACES
Swing

- The dialogs shown are so-called *modal windows*: they need to be closed (by pressing one of the options buttons) before the program can continue and before the user can use any of the other windows of the program.
- Swing also provides
  - simple dialogs for entering text
  - simple dialogs for selecting items from a drop-down list
  - file selection dialogs
  - non-modal dialogs
  - …

JAVA2D

- So far, Swing has painted things like windows and buttons for us
- But Java can of course also be used to paint programmatically
- It can also be used to paint Swing and AWT components (overriding the default looks)
- *Java2D* is an API for two-dimensional graphic operations:
    - Drawing geometric shapes such as lines, circles, rectangles…
    - Filling such shapes with colors or textures
    - Showing images
    - …
- Java2D is part of the AWT and as such part of the Java Standard Edition

JAVA2D

- (Lots) more about Java2D can be found at https://docs.oracle.com/javase/tutorial/2d/

- There is also *Java3D* (not part of the JDK): http://www.oracle.com/technetwork/articles/javase/index-jsp-138252.html

JAVA2D

- Here is a simple example, where Java2D is used together with Swing components.
  In case you want to try out Java2D, you could use this program as a starting point.
- Remark: the code also shows how to override the painting methods of Swing components (in case you are not satisfied by the way Swing paints buttons, lists, tables, etc).
- Note that, like Swing, most of Java2D is not thread-safe. But consequences of multithreading errors are typically less severe, because there are no data models in Java2D, so it's less likely to accidentially mutate any shared resources (shared data)...

JAVA2D

```
import javax.swing.*;

import java.awt.*;

import java.awt.event.*;


class GraphicTest {

   public static void main(String[] args) {
   // so far this is just ordinary Swing code:

        JFrame frame = new JFrame("My window");

        frame.setSize(400, 300);

        JPanel content = new MyJPanel();

        frame.setContentPane(content);  // specify a container to be used as
          // content pane (alternative to getContentPane().add(...))


          frame.addWindowListener(new WindowAdapter() {

            public void windowClosing(WindowEvent event) { System.exit(0); }
          });


          frame.setVisible(true);

 }
```

JAVA2D

```
public class MyJPanel extends JPanel {  // panel ≈ empty space Swing container

  @Override
   public void paintComponent(Graphics g) { // called by Swing to paint the panel

     Graphics2D graphics = (Graphics2D)g; // this Graphics object is used
     // for rendering the visual contents of a Swing component.
     // Graphics2D is the Java2D replacement for the old AWT Graphics class

     Line2D line = new Line2D.Double(20, 20, 160, 160);
       // ".Double" means that the coordinates could be given as double values

      graphics.setColor(Color.black);

      graphics.setStroke(new BasicStroke(10));  //thickness

      graphics.draw(line);
```

JAVA2D

```
// (continued from previous page)

    graphics.setColor(Color.DARK_GRAY);

    Rectangle2D square = new Rectangle2D.Double(10, 10, 300, 120);

    graphics.draw(square);  // draws the outline

    graphics.setPaint(new GradientPaint(0, 0, Color.red, 150, 25, Color.green,
   true));

    graphics.fill(square);  // fills and paints
```

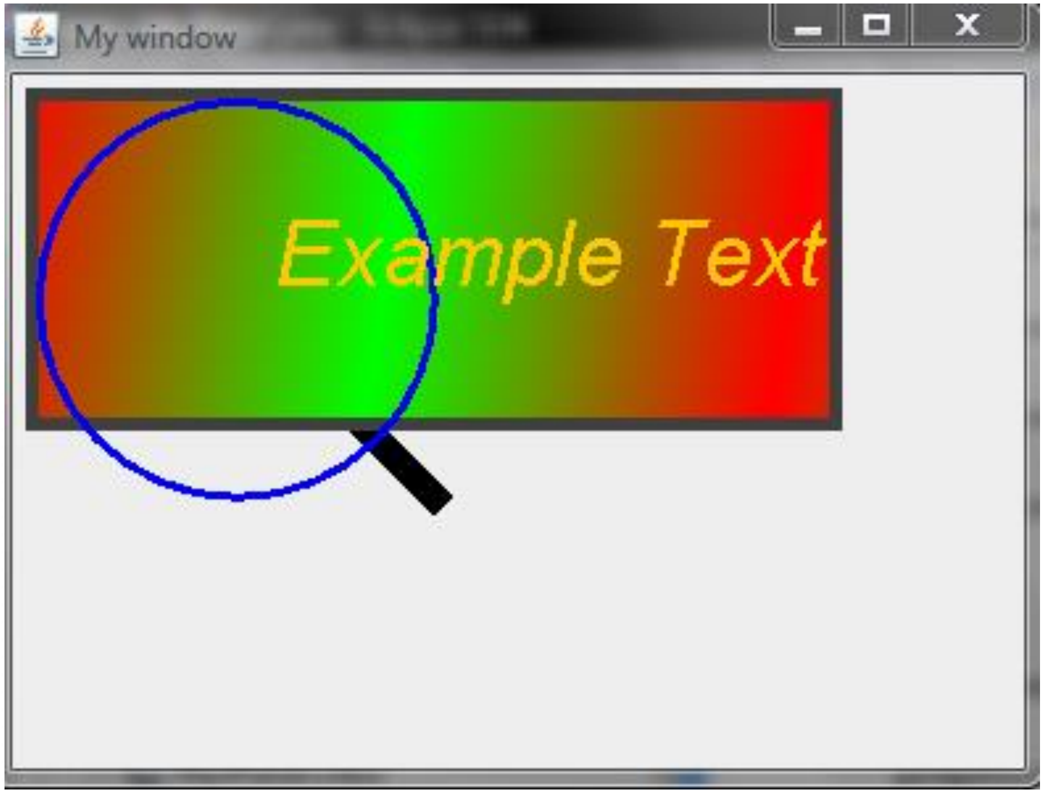
JAVA2D

```
// (continued from previous page)

  Ellipse2D circle = new Ellipse2D.Double(10, 10, 150, 150);

  graphics.setStroke(new BasicStroke(3));

  graphics.setColor(Color.blue);

  graphics.draw(circle);
```

JAVA2D

```
// (continued from previous page)

    FontRenderContext frc = graphics.getFontRenderContext();

    Font f = new Font("Helvetica",Font.ITALIC, 34);

    String s = new String("Example Text");

    TextLayout tl = new TextLayout(s, f, frc);

    graphics.setColor(Color.ORANGE);

    tl.draw(graphics, 100, 80);

  }

}
```

JAVA2D



USER INTERFACES
Model-View-Controller pattern

- Swing and most other modern GUI APIs, but also other kinds of technologies (such as *Servlets*) can be programmed according to the *Model-View-Controller* pattern (MVC).
- The MVC aims at a decoupling the application data and its logic *("model")* on the one hand side, and the visual data presentation ("*view*") on the other hand. Both are glued together by a *controller*.
- This makes it easier to reuse classes and to modify single classes without having to change everything else too. E.g.,
  - the same data model could be visualized by different "views", even at the same time (e.g., a list of numbers shown as a bar graph and as a table)
  - new views can be developed without having to modify the underlying data operations

USER INTERFACES
Model-View-Controller pattern

- The *model* (a.k.a. *data model* - see previous model) provides the domain-specific representation of application data, including methods which operate on this data.
- A *view* renders the model into a visual form suitable for interaction with the user.
- Whenever the data model changes and the user shall be informed, the view is refreshed in order to reflect the modified model.
- The internal communication between model and view is managed by one or more *controller(s)*.

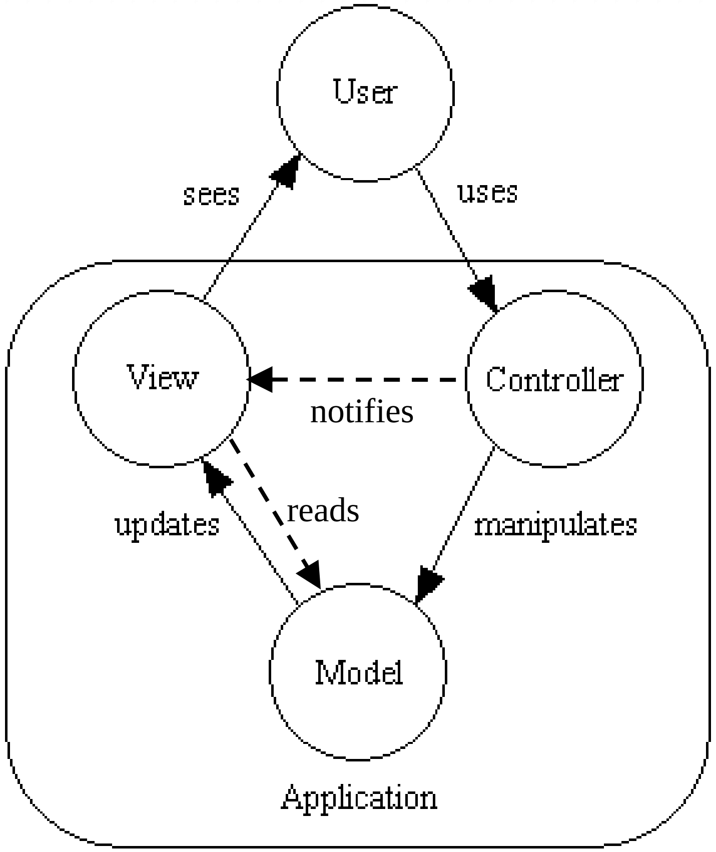
USER INTERFACES
Model-View-Controller pattern

- A *controller* processes user-generated *events* (e.g., mouse click on a button or keyboard input), and may invoke changes on the model and also on the view (depending on the variant of MVC used).
- The controller modifies the model in response to the event.
- Either then the model notifies the view directly about its modification, or the controller notifies the view about the modified model.
- A controller is just a concept, not necessarily a real entity in the program or memory. *Event handlers* (see previous lecture) can be used to implement controllers.
- There might be multiple controllers for one model/view.

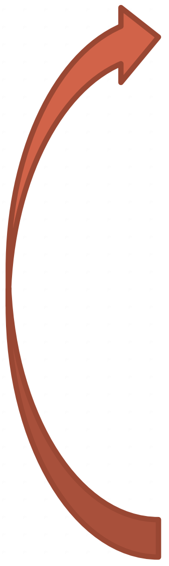
USER INTERFACES

Model-View-Controller pattern

Typical MVC cycle
(variants exist)



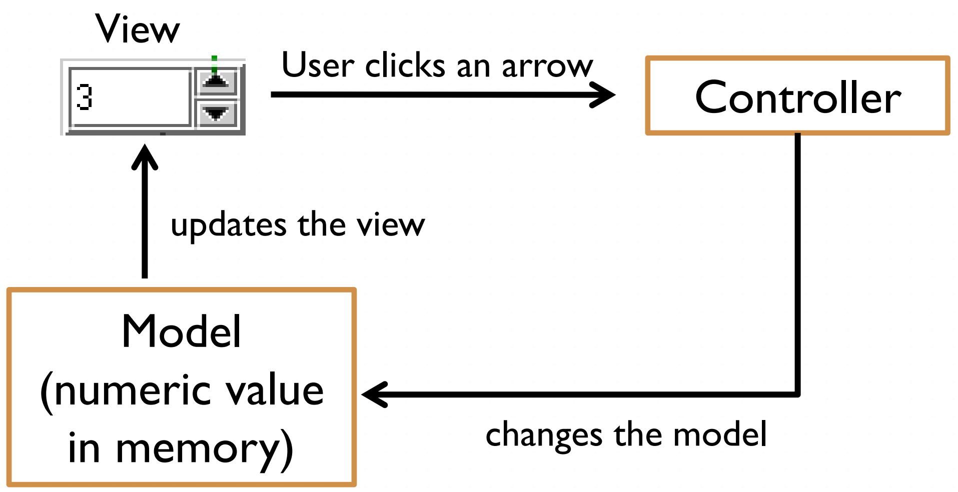
USER INTERFACES
Model-View-Controller pattern



- The user interacts with the user interface (e.g., clicks a button).
- A controller handles this event.
- The controller updates the model in accordance with the event (e.g., computes a number, adds an object to some array, …). Either then the model directly notifies the view or the controller notifies the view about the updated model.
- The view generates from the model an appropriately updated user interface or visual data presentation (e.g., it shows the computed number or the updated content of the array, using GUI components).

USER INTERFACES
Model-View-Controller pattern

Example:



USER INTERFACES
Model-View-Controller pattern

- Different variants of MVC exist - there is no "standard MVC"...
- Some MVC-based GUI frameworks (including Swing!) do not represent model, view, and controller as separate entities. Sometimes, the controller and the view, or the controller and the model, are combined into a single entity.
- In Swing, user actions like clicking or typing cause that GUI components like `JButton` create events. The respective event handler (~controller) then modifies the component's data model appropriately.
- This modification of the model automatically creates another event which causes Swing to update the visual component (so that it displays the modified data). E.g., notification of a `JList` or `JTable` about modified list/table data.

USER INTERFACES

Model-View-Controller pattern

- The MVC pattern can be implemented manually, but GUIs like Swing also provide built-in support for (simplified) MVC.
- For example, `Jlist`, `JTable` and `JTree` separate the data (“model”) and their visible representation, and use listeners (event handlers) as controllers.
- But simpler Swing components can have models too (e.g., the text in a text field or the state of a button).

- The following example shows how to create a `JList` component using a data model...

USER INTERFACES
Model-View-Controller pattern

```
DefaultListModel listModel = new DefaultListModel();
 listModel.addElement("1");
 listModel.addElement("22");
 listModel.addElement("333");

 JList myJList = new JList(listModel);
 contentPane.add(myJList, BorderLayout.SOUTH);
      ...
listModel.remove(1);  // here we change the model. E.g., as a result of
                      // pushing a button.
     // The "view" (i.e., the visual component myJList) is notified
     // automatically by Swing about this modification to the model
     // using a list data event which is internally handled
```

USER INTERFACES
Model-View-Controller pattern

- Instead of copying the data directly into the `JList` (the view), you tell the `JList` to use a separate data list (the model).
- Changes of the data model lead to changes of presentation of the list on the screen, i.e., the view.
- Changes are communicated via data change events (see previous lecture)
- Multiple `JLists` could even share the same data (i.e., the same model).
- The actual model here is the `DefaultListModel` object (under the hood a `java.util.Vector`).
- There are no dedicated controller objects in Swing. The controllers exist only *implicitly* as a combination of GUI component functionality and event handlers. However, sometimes programmers create their own controller classes and methods.

USER INTERFACES
Model-View-Controller pattern

- `JTable` components are also a good example for how Swing implements a version of the MVC paradigm.
- Instead of a "default model" for the table data (which also exists in Swing), we use again our own data model class, derived from `AbstractTableModel` (cf. previous lecture). It allows to specify the methods which Swing will invoke at runtime to *dynamically* compute the cell contents of the table, the number of rows and columns, etc.
- See next slides. Similar to the JTable example code from the previous lecture, but now as a full program...

USER INTERFACES
Model-View-Controller pattern

```
import java.awt.*;
import javax.swing.*;
import javax.swing.table.AbstractTableModel;

class TableDemoFrame extends JFrame {

    private JTable table;

    TableDemoFrame() {

        setTitle("Demo frame with a JTable component");

        setSize(300, 300);
```

USER INTERFACES

Model-View-Controller pattern

The *model* in MVC

```
AbstractTableModel dataModel = new AbstractTableModel() { // anonymous class
    public int getColumnCount() {
            return 4;
    }

    public String getColumnName(int column) {
            return "Column-" + column;  // column headers
    }

    public int getRowCount() {
            return 20;
    }
    public Object getValueAt(int row, int col) {
            return new Integer(row*col); // cell could contain any Java object
              //    ^ the cell content at coordinates (row, column)
    }
};
```

name of the abstract class or interface implemented by our anonymous class

USER INTERFACES

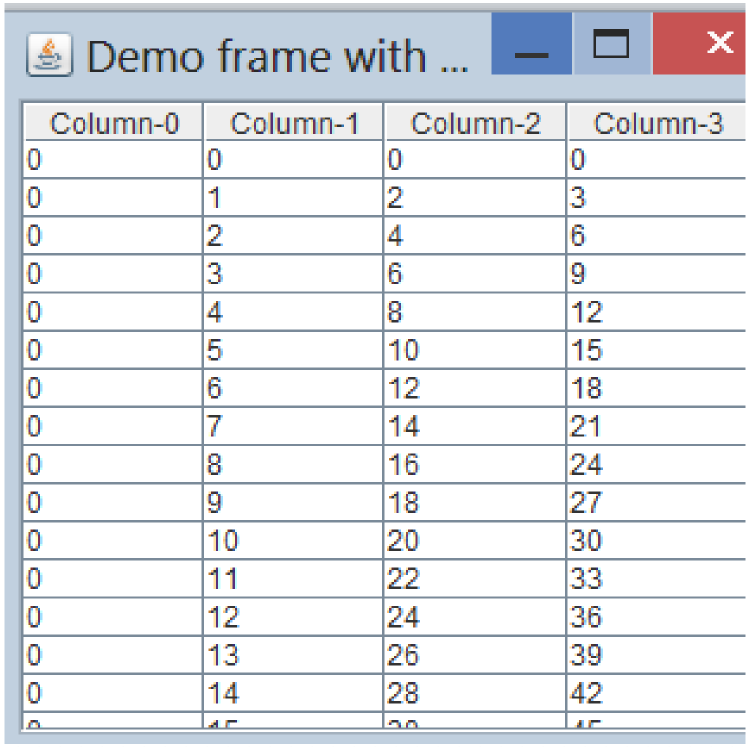
Model-View-Controller pattern

The *view* in MVC

```
      table = new JTable(dataModel);

      getContentPane().setLayout(new BorderLayout());

      getContentPane().add(table, BorderLayout.CENTER);

    }

    public static void main(String args[]) {

       TableDemoFrame frame = new TableDemoFrame();

       frame.setVisible(true);

     }
 }
```

USER INTERFACES

Model-View-Controller pattern



Column-0	Column-1	Column-2	Column-3
0	0	0	0
0	1	2	3
0	2	4	6
0	3	6	9
0	4	8	12
0	5	10	15
0	6	12	18
0	7	14	21
0	8	16	24
0	9	18	27
0	10	20	30
0	11	22	33
0	12	24	36
0	13	26	39
0	14	28	42

NESTED CLASSES

- We have already introduced anonymous classes. They are a form of more general concepts (partially known from Semester 1):
  nested classes and *inner classes*
- *Nested classes* are classes which are declared inside of other classes. They can be static or non-static.

```
class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
    class InnerClass {
        ...
    }
}
```

- *Inner classes* are <u>non-static nested classes</u>
- More on nested classes:
  https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html

NESTED CLASSES

- Remark: terminology is inconsistent. Sometimes only *static* nested classes are called "nested classes". Sometimes nested classes are called "member classes".

NESTED CLASSES

- Purposes of nested classes:
  - *Grouping of classes which belong together*: If a class is useful for only one other class, then these two should be kept together.
  - *Avoidance of extra files*. Nested classes don’t require their own source files (but the compiler creates extra class files (those with a $ sign in it)…).
  - *Improved encapsulation*: By hiding a class B within a class A, A's members can be declared private but (only) B can nevertheless access them.

NESTED CLASSES

- *Static nested classes* are members(!) of the outer class and can be declared public, private, package-visible ("default") or protected, just like fields or methods.
- They can access only static members of the outer class directly (i.e., without creating an object of the outer class). They can even access *private* static members of the outer class.
- Outside the outer class, static nested classes are accessible with `OuterClass.StaticNestedClass`
- All in all, static nested classes behave more or less like ordinary (i.e., top-level) classes, with the outer class playing a roughly similar role to the nested class as a package to a non-nested class.

```java
import java.util.Comparator;

class CompareTestInner1
{
    private static class OrderRectByWidth implements Comparator<SimpleRectangle>
    {
        public int compare( SimpleRectangle r1, SimpleRectangle r2 )
          { return r1.getWidth( ) - r2.getWidth( ); }
    }

    public static void main( String [ ] args )
    {
        SimpleRectangle [ ] rects = new SimpleRectangle[ 4 ];
        rects[ 0 ] = new SimpleRectangle( 1, 10 );
        rects[ 1 ] = new SimpleRectangle( 20, 1 );
        rects[ 2 ] = new SimpleRectangle( 4, 6 );
        rects[ 3 ] = new SimpleRectangle( 5, 5 );

        System.out.println( "MAX WIDTH: " +
             Utils.findMax( rects, new OrderRectByWidth( ) ) );
    }
}
```

creates an object (more precisely, a functor) of the static nested class OrderRectByWidth

NESTED CLASSES

- In contrast to static nested classes, *inner classes* are associated with an instance (object) of the outer class.
- Except for -->*local* and *anonymous* classes (which are both also forms of inner classes), they are declared like static nested classes, but without the `static` qualifier.
- An object of the inner class exists in some sense *"within" an object of the outer class*.
- Creating an instance of the inner class thus requires an object of the outer class. The "inner object" becomes embedded in the "outer object".
  The - somewhat strange - Java syntax for this is:

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

NESTED CLASSES

- Inside of code in the inner class (i.e., within methods of the inner class), the outer object "around" the instance of the inner class can be referenced using `OuterClass.this`
- If there is no danger of a name clash, the `OuterClass.this` can be omitted when accessing members of the outer class inside of the inner class. The inner class method can then use a field or method of the outer class object directly, just like it can (of course) use members of the inner class.
- As an example, the following slides show two alternative implementations of container iterators (see lecture about collections): first without inner classes and afterwards using inner classes.

```java
package weiss.ds;

public interface Iterator
{
    boolean hasNext( );
    Object next( );
}
```

Examples from Mark Allen Weiss: Data Structures and Problem Solving Using Java

An implementation of Iterator *without* using an inner class. The container accessed by the iterator is here simply a field (`private MyContainer container`):

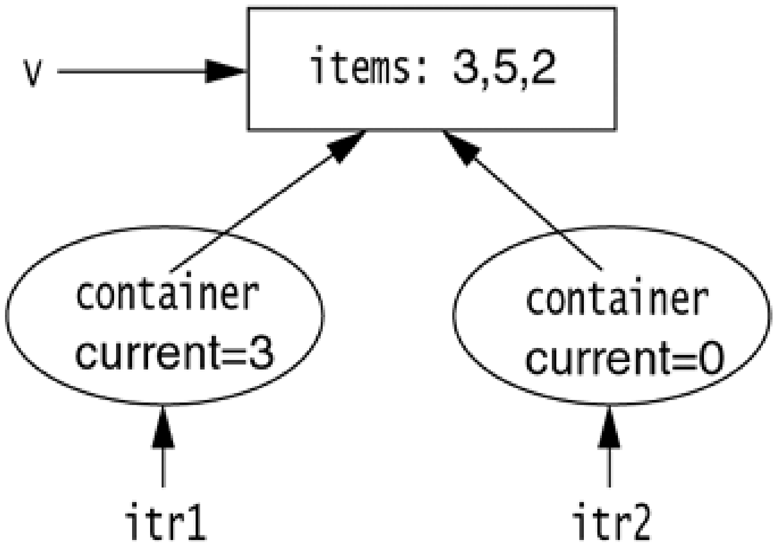
```java
// An iterator class that steps through a MyContainer.

package weiss.ds;

class MyContainerIterator implements Iterator
{
    private int current = 0;
    private MyContainer container;

    MyContainerIterator( MyContainer c )
      { container = c; }

    public boolean hasNext( )
      { return current < container.size; }

    public Object next( )
      { return container.items[ current++ ]; }
}
```

The container (e.g., list) this iterator iterates over.

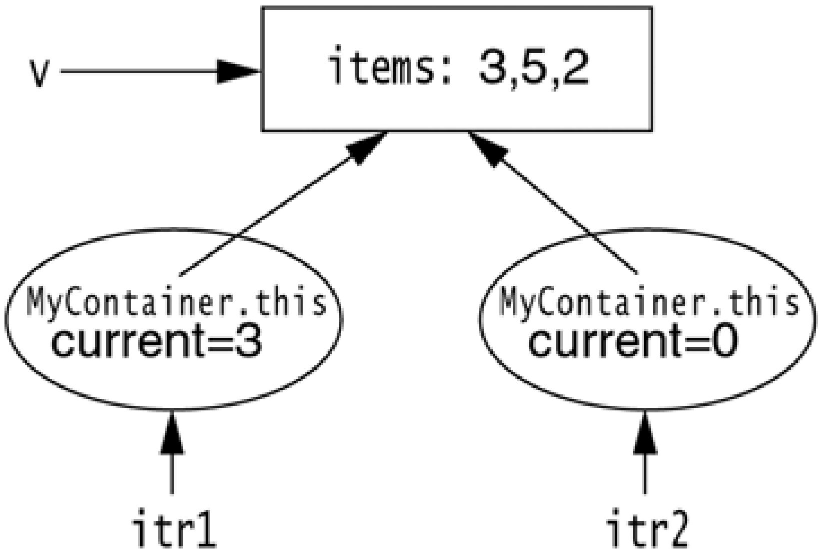
```java
package weiss.ds;

public class MyContainer
{
    Object [ ] items;
    int size;

    public Iterator iterator( )
      { return new MyContainerIterator( this ); }

    // Other methods not shown.
}
```

```java
public static void main( String [ ] args )
{
    MyContainer v = new MyContainer( );

    v.add( "3" );
    v.add( "2" );

    System.out.println( "Container contents: " );
    Iterator itr = v.iterator( );
    while( itr.hasNext( ) )
        System.out.println( itr.next( ) );
}
```

Every iterator instance references a single container:



More elegant: Using an inner class instead for the iterators allows us to put the iterator object directly <u>inside</u> of the container object…



```java
package weiss.ds;

public class MyContainer
{
    private Object [ ] items;
    private int size = 0;

    // Other methods for MyContainer not shown

    public Iterator iterator( )
      { return new LocalIterator( ); }

    // The iterator class as an inner class
    private class LocalIterator implements Iterator
    {
        private int current = 0;

        public boolean hasNext( )
          { return current < MyContainer.this.size; }

        public Object next( )
          { return MyContainer.this.items[ current++ ]; }
    }
}
```

"outer class"

inner class
LocalIterator

`"Outer.this"` can be omitted (if there is no name ambiguity):

```
// The iterator class as an inner class
private class LocalIterator implements Iterator
{
    private int current = 0;

    public boolean hasNext( )
      { return current < size; }

    public Object next( )
      { return items[ current++ ]; }
}
```

NESTED CLASSES

- The benefits for using an inner class in the previous examples:
  - Fields in `MyContainer` can be made private: improved encapsulation.
  - Field `container` in the iterator can be omitted: the inner class object always knows its outer class object (here: container).
  - Use of the inner class expresses elegantly that iterators are only and always used together with containers.

  => Improved organization of code, better maintainability, more compact

NESTED CLASSES

- *Local classes (*also called *local inner classes)* are inner classes which are declared *within a code block (e.g., a* method body) in the outer class.
- Whereas ordinary inner classes can be visible outside of the outer class if not declared private, local classes are always visible only within the code block where they are declared.
- Just like ordinary inner classes, they can access the members of the outer class.
- Additionally, they can access method parameters and local variables which have been declared in the method (where the local class is located) prior to the declaration of the local class, provided these variables have been declared `final`.

NESTED CLASSES

Example:

```
public class OuterClass {
   void oMethod() {
        int var = 0;
        final int fvar = 1;
        class LocalClass {  // a local class
           public void iMethod() {
               int i = fvar;  // fine since fvar is final
               // int i2 = var;  would be illegal since var is not final
           }
        }; // observe the ;
       new LocalClass().iMethod();
    }
}
```

NESTED CLASSES

- Anonymous classes (a.k.a. *anonymous inner classes*) are simply local classes without a name
  → see previous lecture for details
- Often used to implement event handlers and functors
- With Java 8 or higher, consider *Lambda Expressions* as an alternative to anonymous classes for the purpose of defining functors (→ one of the next lectures)

Yet another example: a functor for comparing geometric figures, implemented using an anonymous class which implements a generic interface:

```
class CompareTestInner3
{
    public static void main( String [ ] args )
    {
        SimpleRectangle [ ] rects = new SimpleRectangle[ 4 ];
        rects[ 0 ] = new SimpleRectangle( 1, 10 );
        rects[ 1 ] = new SimpleRectangle( 20, 1 );
        rects[ 2 ] = new SimpleRectangle( 4, 6 );
        rects[ 3 ] = new SimpleRectangle( 5, 5 );

        System.out.println( "MAX WIDTH: " +
            Utils.findMax( rects, new Comparator<SimpleRectangle>( )
            {
                public int compare( SimpleRectangle r1, SimpleRectangle r2 )
                  { return r1.getWidth( ) - r2.getWidth( ); }
            }
        ) );
    }
}
```