AOT LAB

**A**gent and **O**bject **T**echnology **Lab**
Dipartimento di Ingegneria dell’Informazione
Università degli Studi di Parma

# Standard Widget Toolkit and JFace

**Alessandro Negri**

negri@ce.unipr.it

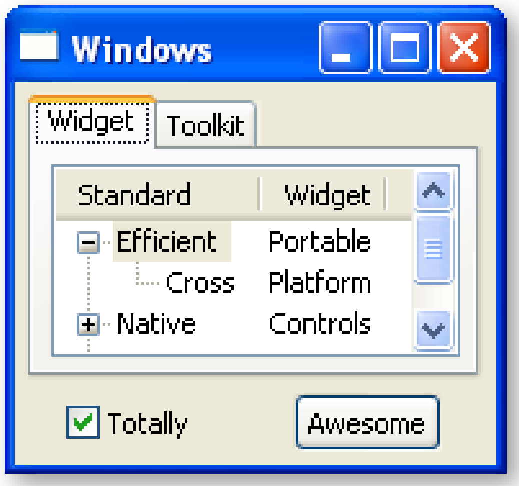
http://www.ce.unipr.it/people/negri/

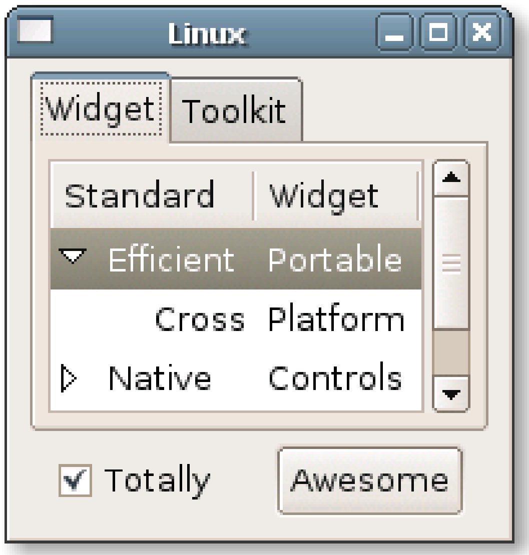
AOT
LAB

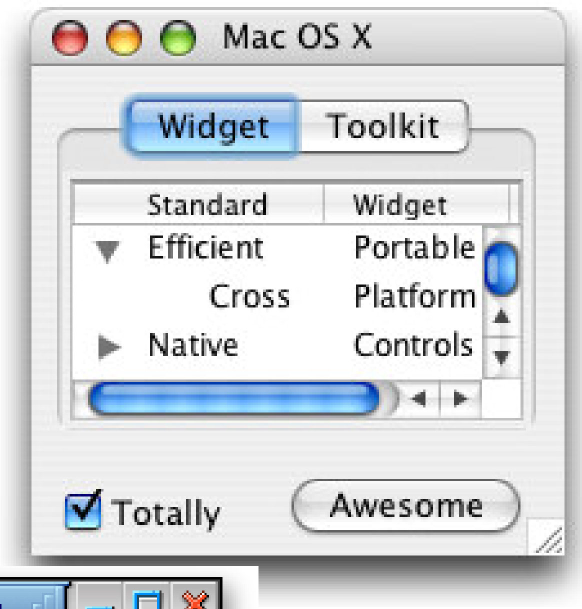
## Standard Widget Toolkit

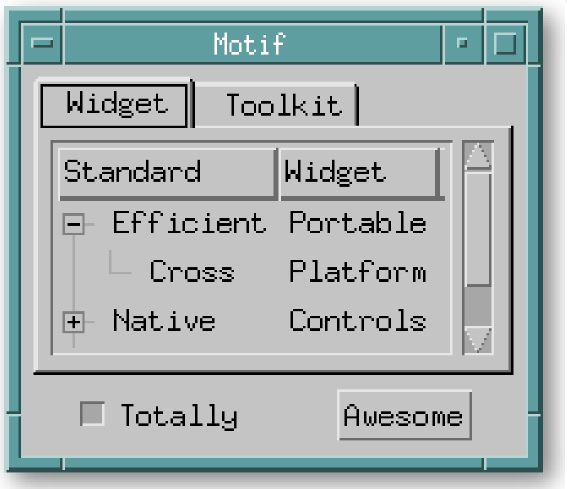
- Graphical User Interface library to help you build indipendent desktop applications
  - Buttons, lists, text, menu, trees…
- OS indipendent APIs, but…
- … it needs specific components for each OS
  - On Windows as a.dll, on Linux as a .so
- Every graphical project in Eclipse must use SWT, but …
- … SWT maybe used also indipendently from Eclipse
  - You need only to include in your classpath the file: swt.jar
  - SWT vs Swing…

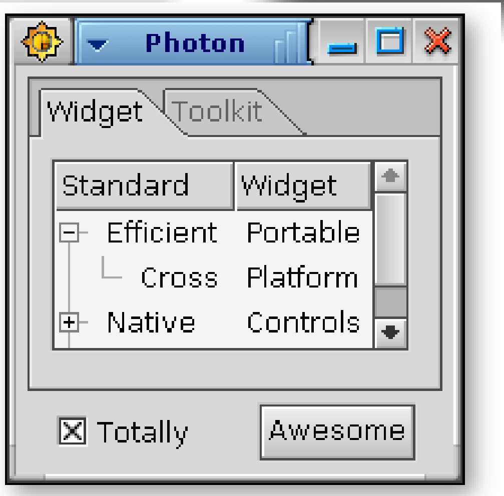
## SWT - Examples





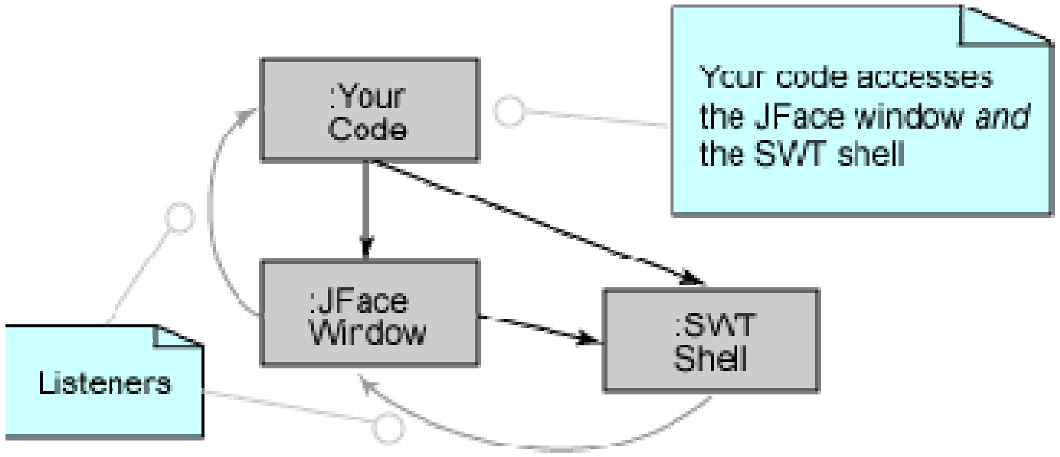






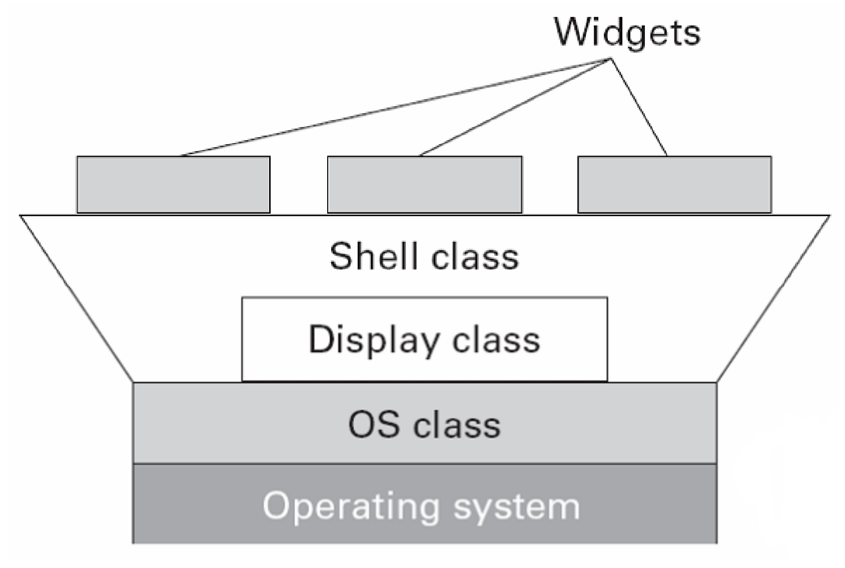
## JFace

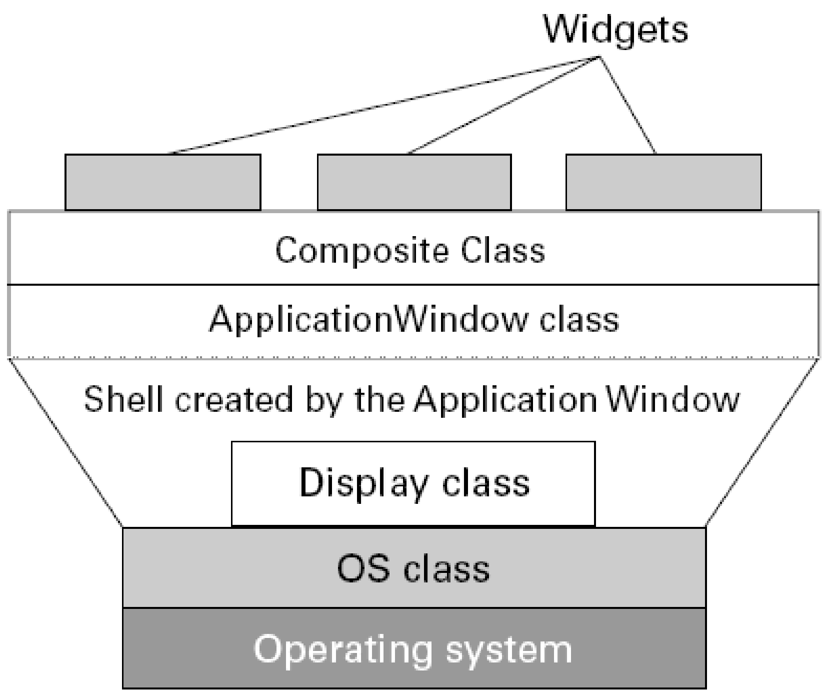
- JFace is a UI toolkit built on top of SWT
- It uses SWT widgets to implement GUI code that is common among different applications
  - Using JFace, you can create user interface components with less code than if you had started at the basic SWT widget level
- It is not really a layer on top of SWT and it doesn't try to hide SWT from you
  - It recognizes that there are several common patterns of use for SWT and it provides utility code to help you program these patterns more easily



AOT
LAB

## SWT vs JFace





AOT
LAB

## Hello World

```
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.SWT;

public class HelloWorld {
   public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        Label label = new Label(shell, SWT.CENTER);
        label.setText("Hello, World");
        label.setBounds(shell.getClientArea());
        shell.open();
        while (!shell.isDisposed()) {
          if (!display.readAndDispatch()) {
                display.sleep();
          }
        }
        display.dispose();
        }
}
```

Represents the underlying windowing system

AOT LAB

## Hello World

```
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.SWT;

public class HelloWorld {
   public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        Label label = new Label(shell, SWT.CENTER);
        label.setText("Hello, World");
        label.setBounds(shell.getClientArea());
        shell.open();
        while (!shell.isDisposed()) {
          if (!display.readAndDispatch()) {
                display.sleep();
          }
        }
        display.dispose();
        }
}
```

It is an abstraction that represents a top-level window when created with a Display object

AOT
LAB

## Hello World

The Label object is capable of displaying either simple text, as you use it here, or an image. The widget is constructed with a reference to a Shell object, which is an indirect descendant of the Composite class

```
in
in
in
in

pu

        Display d            new Display();
        Shell shell = ne    ell(display);
        Label label = new Label(shell, SWT.CENTER);
        label.setText("Hello, World");
        label.setBounds(shell.getClientArea());
        shell.open();
        while (!shell.isDisposed()) {
          if (!display.readAndDispatch()) {
                display.sleep();
          }
        }
        display.dispose();
        }
}
```

AOT
LAB

## Hello World

```
import org.eclipse.swt.widgets.Display;
:
:
:
:

p


        Shell               hell(display);
        Label               Label(shell, SWT.CENTER);
        label.s        ("Hello, World");
        label.se     ounds(shell.getClientArea());
        shell.open();
        while (!shell.isDisposed()) {
          if (!display.readAndDispatch()) {
                display.sleep();
          }
        }
        display.dispose();
        }
}
```

Indicates to the underlying system to set the current shell visible, set the focus to the default button (if one exists) and make the window associated with the shell active. This displays the window and allows it to begin receiving events from the underlying windowing system

## Hello World

```
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.SWT;

public class HelloWorld {
   public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        Label label = new Label(shell, SWT.CENT
        label.setText("Hello, World");
        label.setBounds(shell.getClientArea());
        shell.open();
        while (!shell.isDisposed()) {
          if (!display.readAndDispatch()) {
                display.sleep();
          }
        }
        display.dispose();
        }
}
```

Main Window Loop

AOT
LAB

## Hello World

```
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.SWT;

public class HelloWorld {
   publi
        D
        S
        L
        l
        l
        shell.open
        while (!sh          posed()) {
          if (!dis          dAndDispatch()) {
                dis       leep();
          }
        }
        display.dispose();
        }
}
```

Because your window has been disposed (by the user closing the window), you no longer need the resources of the windowing system to display the graphical components

**DEMO**

## Control & Composite Classes

- A ***Control*** is a user interface element that is contained somewhere within a top-level window, the ***Shell***
- Controls are common in all user interfaces
  - Buttons, labels, progress bars, and tables are all controls
- Bottom Up view
  - Every control has a parent that is an instance of the class ***Composite*** or one of its subclasses.
  - The class Shell is a subclass of Composite. Shells are created on a ***Display***, which represents the "screen"
- Top Down view
  - A ***Display*** contains a list of top-level Shells, where each Shell is the root of a tree composed of Composites and Controls
  - Composites can contain other Composites, allowing the tree to have arbitrary depth. If the child of a Shell is another Shell, the child is commonly called a ***Dialog Shell***
    - A Dialog Shell always stays in front of the parent Shell

## Widgets?

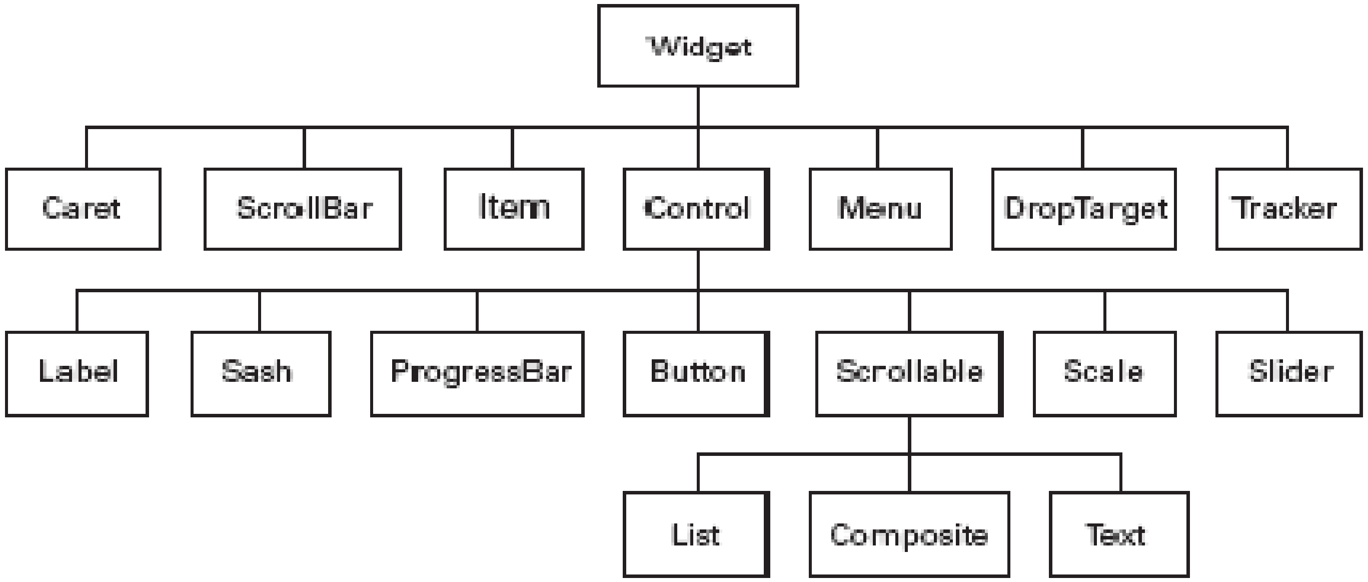
- A widget is a graphical user interface element responsible for interacting with the user
- Widgets maintain and draw their state using some combination of graphical drawing operations
- Using the mouse or the keyboard, the user can change the state of a widget
  - When a state change occurs, whether initiated by the user or the application code, widgets redraw to show the new state
  - It means that when you set a property on a widget, you are not responsible for telling the widget to redraw to reflect the change

## Widget Class

- Abstract superclass for all UI object
- Created using constructors (no "factories")
  - It unifies all widgets under one structure
- OS resources are allocated when a widget is created
  - Disposed by the user or programmatically with dispose()
  - OS resources are released when a widget is disposed
- Notifies listeners when widget-specific events occur
- Allows application-specific data to be stored
  - if a widget must be shared across different classes and must contain information beyond that normally provided by its class
  - if a widget has a global scope and must provide information across procedures that can’t directly communicate with each other

AOT LAB

## Widgets Hierarchy



1. Widget ()
2. Widget (Widget parent)
3. Widget (Widget parent, int style)
4. Widget (Widget parent, int style, int index)

- Widgets cannot exist without a parent, and the parent cannot be changed after a widget is created

AOT LAB

## Managing Operating System Resources

- SWT uses operating system resources to deliver its native graphics and widget functionality
- Languages that include garbage collection, such as the Java™ language, relieve the programmer from the burden of managing memory
  - **but not from the allocation and freeing of operating system resources**

```
Color blue = new Color (display, 0, 0, 255);
blue.dispose();
```

AOT LAB

## Example: dispose()

```
import org.eclipse.swt.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class Broken {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setLayout(new RowLayout());
        Text text = new Text(shell, SWT.BORDER);
        shell.open();
        while (!shell.isDisposed()) {
          if (!display.readAndDispatch()) {
                display.sleep();
          }
        }
        System.out.println(text.getText());
        display.dispose();
    }
}
```

DEMO

## dispose()

- Hides the widget and its children
  - releases all associated operating system resources
- Removes the widget from the children list of its parent
  - All references to other objects in the widget are set to null, facilitating garbage collection
- Accessing a widget after it has been disposed is an error and causes an SWTException ("Widget is disposed") to be thrown
- When a widget is disposed of, a dispose event is sent, and registered listeners are invoked in response
  - see later…

## Dispose Rules

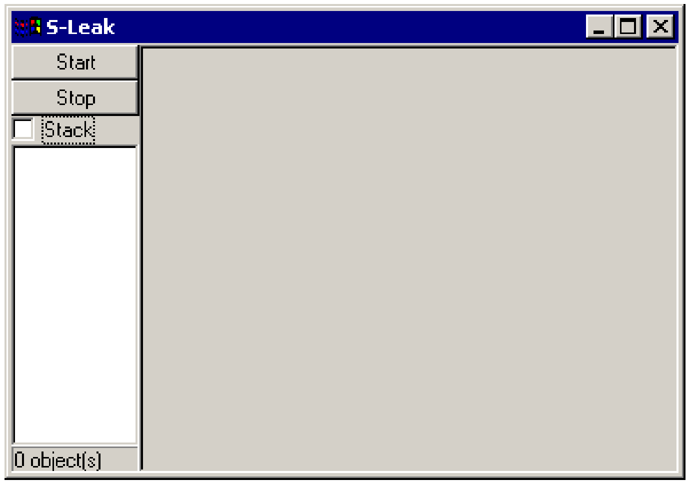
- **RULE 1**: If you created it, you dispose of it
  - SWT ensures that all operating system resources are acquired when the widget is created (in the constructor method)
  - You are responsible for calling dispose() on SWT objects that you created using new
  - SWT will never create an object that needs to be disposed by the programmer outside of a constructor
- **RULE 2**: Disposing a parent disposes the children
  - Disposing of a top-level shell will dispose its children
  - Disposing of a menu will dispose its menu items
  - Disposing of a tree widget will dispose of the items in the tree

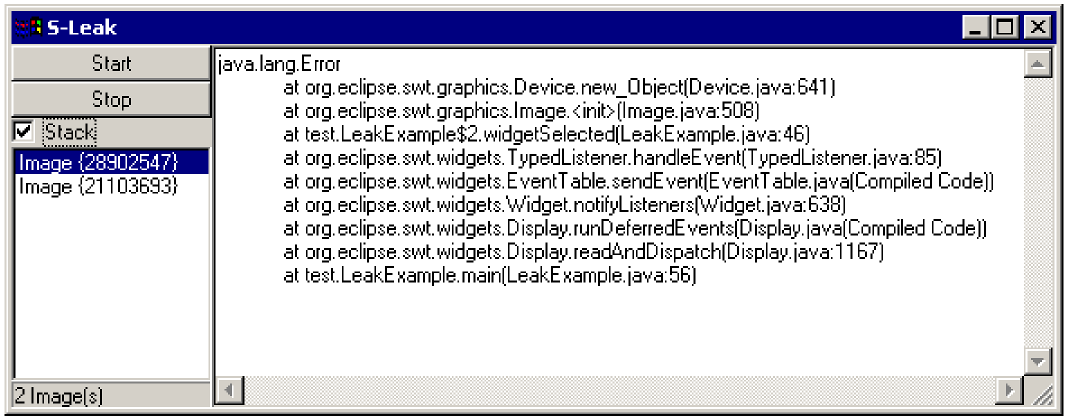
## Dispose: Conclusions

- There are resources that do not have explicit parents, like fonts and colors
  - These must be disposed manually if you created them
    - However, most of the time you will use standard system colors and fonts which you don't have to dispose because they are managed by SWT
- When you still need to create a custom resource (like an icon image), SWT has a class JFaceResources
  - It is a central registry for images, fonts and colors
  - This way you can reuse resources and don't need to worry about their later disposal
- If you absolutely want to dispose a resource manually (e.g. dispose a large image when its containing composite is disposed), do not override dispose() of the containing widget
  - The dispose() method will not be called if the containing parent is disposed
  - Instead add a **DisposeListener** and clean up there

## S-Leak

- Sleak is a simple tool that monitors SWT graphics resources
  - You can use Sleak to detect leaks in SWT application code
- Download:
  - http://www.eclipse.org/articles/swt-design-2/sleak.htm





```
java.lang.Error
	at org.eclipse.swt.graphics.Device.new_Object(Device.java:641)
	at org.eclipse.swt.graphics.Image.<init>(Image.java:508)
	at test.LeakExample$2.widgetSelected(LeakExample.java:46)
	at org.eclipse.swt.widgets.TypedListener.handleEvent(TypedListener.java:85)
	at org.eclipse.swt.widgets.EventTable.sendEvent(EventTable.java(Compiled Code))
	at org.eclipse.swt.widgets.Widget.notifyListeners(Widget.java:638)
	at org.eclipse.swt.widgets.Display.runDeferredEvents(Display.java(Compiled Code))
	at org.eclipse.swt.widgets.Display.readAndDispatch(Display.java:1167)
	at test.LeakExample.main(LeakExample.java:56)
```

AOT LAB

## Class org.eclipse.swt.SWT

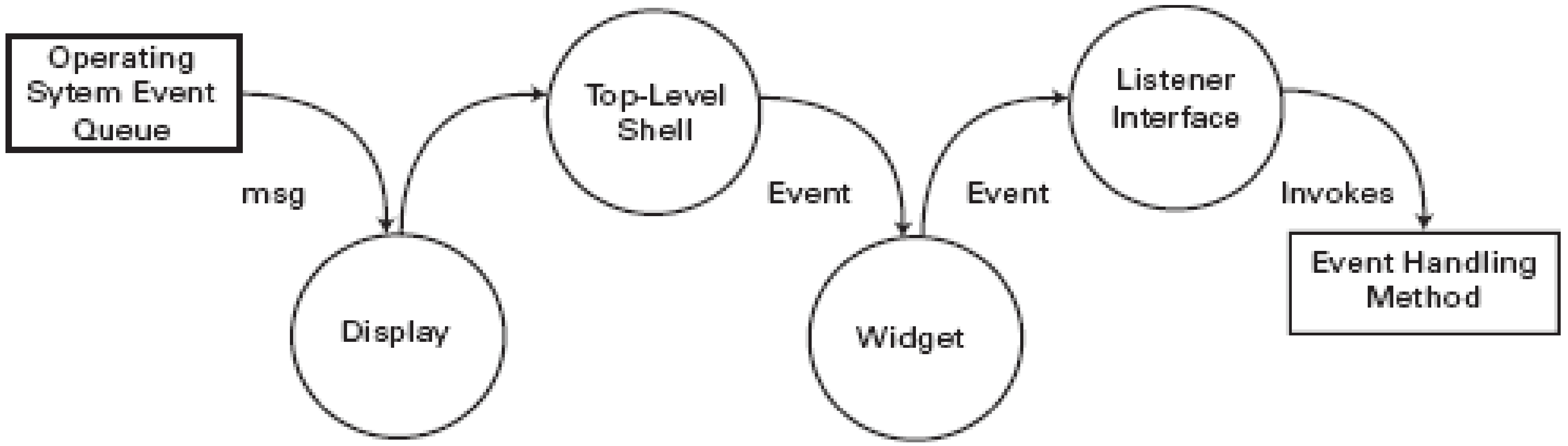
- SWT uses a single class named (appropriately) SWT to share the constants that define the common names and concepts found in the toolkit
  - This minimizes the number of classes, names, and constants that application programmers need to remember
  - The constants are all found in one place

## SWT - Shell Styles

BORDER: Adds a border.
CLOSE: Adds a close button.
MIN: Adds a minimize button.
MAX: Adds a maximize button.
NO_TRIM: Creates a Shell that has no border and can't be moved, closed, resized, minimized, or maximized. Not very useful, except perhaps for splash screens.
RESIZE: Adds a resizable border.
TITLE: Adds a title bar.
DIALOG_TRIM: Convenience style, equivalent to TITLE | CLOSE | BORDER.
SHELL_TRIM: Convenience style, equivalent to CLOSE | TITLE | MIN | MAX | RESIZE.
APPLICATION_MODAL: Creates a Shell that's modal to the application. Note that you should specify only one of APPLICATION_MODAL, PRIMARY_MODAL, SYSTEM_MODAL, or MODELESS; you can specify more, but only one is applied. The order of preference is SYSTEM_MODAL, APPLICATION_MODAL, PRIMARY_MODAL, then MODELESS.
PRIMARY_MODAL: Creates a primary modal Shell.
SYSTEM_MODAL: Creates a Shell that's modal system-wide.
MODELESS: Creates a modeless Shell.

AOT LAB

## Event Handling



- **Event**
  - An indication that something interesting has happened
  - Event classes, used to represent the event, contain detailed information about what has happened

## Listeners

- Listener
  - Instance of a class that implements one or more agreed-upon methods whose signatures are captured by an interface
  - Listener methods always take an instance of an event class as an argument
    - When something interesting occurs in a widget, the listener method is invoked with an appropriate event
- Two types of listeners
  - Unityped
  - Typed

## Untyped Listeners

- Simple, generic, low-level mechanism to handle any type of event
- There are only two Java types involved:
  - a single generic interface called Listener
  - a single event class called Event
- Untyped listeners are added using the method **addListener**()
  - Adds the listener to the collection of listeners that will be notified when an event of the given type occurs
  - When the event occurs in the widget, the listener is notified by calling its **handleEvent**() method

AOT
LAB

## Esempio: addListener()

```
widget.addListener(SWT.Dispose, new Listener()  {
    public void handleEvent(Event event) {
            // widget was disposed
    }
});
```

- When multiple listeners are added, they are called in the order they were added
  - This gives the first listener the opportunity to process the event and possibly filter the data **before** the remaining listeners are notified
  - Adding the same instance of a listener multiple times is supported, causing it to be invoked once for each time it was added

## Typed Listeners

- Follow the standard “JavaBeans listener pattern”
- Typed listeners and their corresponding event classes are found in the package **org.eclipse.swt.events**
  - To listen for a dispose event on a widget, application code would use addDisposeListener()
    - Adds the listener to the collection of listeners that will be notified when a widget is disposed
    - When the widget is disposed, the listener is notified by calling its widgetDisposed() method

AOT
LAB

## Esempio: Dispose Listener

```
widget.addDisposeListener(new DisposeListener() {
    public void widgetDisposed(DisposeEvent event) {
            // widget was disposed
    }
});
```

- Dispose Listener is notified on the associated widget's disposal event

AOT
LAB

## Typed vs Untyped Listeners

- Though less friendly to code with, untyped listeners can lead to smaller, though potentially uglier, code
- You can also freely add listeners to widgets for events that the widgets don't support
  - The compiler won't complain, and the program won't throw exceptions at run time
- Typed listeners lead to more modular designs and also make clear which events a particular widget supports

- **Either typed or untyped widgets work equally well in running code**

AOT
LAB

## Layouts

- A **Layout** is an instance of a class that implements a positioning and sizing algorithm
- Layouts are used to arrange controls within a composite automatically
- Class Layout is not a subclass of class Widget, so it shares none of the Widget behavior
  - There is no operating system storage associated with a layout
  - Instances do not need to be sent the dispose() message when you are finished with them
- Why use Layouts?
  - Ability to compute a good initial size of a composite
  - Automatic repositioning of child controls when a composite is resized
  - Operating system and locale (language)-independent positioning and sizing
  - Maintenance of layout code is often easier than the equivalent positioning and sizing code

AOT
LAB

## Layout Data

- There are two ways to control how an instance of a particular Layout class behaves
  - Each Layout class provides API that is specific to the algorithm that it implements
  - Algorithm-specific layout data can be associated with each child control in the composite
- **Layout data** is any object that a particular layout can use to configure a control
  - The layout data typically holds some form of constraint description that provides additional information about how the control should be arranged
    - The form of this description is completely up to the layout
  - Layouts are associated with composites, whereas layout data is associated with each of the children of the composite
  - For each of the standard SWT Layout classes (that require layout data), there is a specific layout data class
    - GridData, RowData, FormData

AOT
LAB

## FillLayout

```
public static void main(String[] args) {
   Display display = new Display();
   Shell shell = new Shell(display);
   shell.setLayout(new FillLayout());
   Button button = new Button(shell, SWT.PUSH);
   button.setText("Button");
   shell.pack();
   shell.open();
   while (!shell.isDisposed()) {
        if (!display.readAndDispatch())
               display.sleep();
   }
   display.dispose();
}
```

- FillLayout is the simplest layout
  - It fills the available space in a composite with the children of the composite in a single row or column
  - The available space is divided evenly among the children

AOT
LAB

## RowLayout

```
public static void main(String[] args) {
   Display display = new Display();
   Shell shell = new Shell(display);
   shell.setLayout(new RowLayout(SWT.VERTICAL));
   for (int i=0; i<12; i++) {
        Button button = new Button(shell, SWT.PUSH);
        button.setText("B" + i);
   }
   shell.pack();
   shell.open();
   while (!shell.isDisposed()) {
        if (!display.readAndDispatch())
               display.sleep();
   }
   display.dispose();
}
```

- RowLayout positions controls in either rows or columns but, unlike FillLayout, does not fill all the available space in the composite
  - Most often used to position a row/column of buttons

## GridLayout

```
public static void main(String[] args) {
   Display display = new Display();
   Shell shell = new Shell(display);
   Display display = new Display();
   Shell shell = new Shell(display);
   GridLayout gridLayout = new GridLayout();
   gridLayout.numColumns = 3;
   shell.setLayout(gridLayout);
   Button button = new Button(shell, SWT.PUSH);
   . . .
   GridData gridData = new GridData();
   gridData.horizontalAlignment = GridData.FILL;
   gridData.grabExcessHorizontalSpace = true;
   button.setLayoutData(gridData);
   . . .
}
```

- Instances of this class layout the control children of a Composite in a grid

## FormLayout

- **FormLayout** is independent from the complete layout
  - The position and size of the components depend on one Control
- **FormData** is fundamental to set the different layout properties for a widget
  - A FormData object can have 0 or 4 **FormAttachment** objects
    - The form attachment can have controls

AOT
LAB

## FormLayout: Example

```
public static void main(String[] args) {
    . . .
    FormLayout layout= new FormLayout();
    shell.setLayout (layout);

    Button button1 = new Button(shell, SWT.PUSH);
    Button button2 = new Button(shell, SWT.PUSH);
    Button button3 = new Button(shell, SWT.PUSH);
    button1.setText("B1");
    button2.setText("B2");
    button3.setText("B3");

    FormData data1 = new FormData();
    data1.left = new FormAttachment(0,5);
    data1.right = new FormAttachment(25,0);
    button1.setLayoutData(data1);

    FormData data2 = new FormData();
    data2.left = new FormAttachment(button1,5);
    data2.right = new FormAttachment(90,-5);
    button2.setLayoutData(data2);

    FormData data3 = new FormData();
    data3.top = new FormAttachment(button2,5);
    data3.bottom = new FormAttachment(100,-5);
    data3.right = new FormAttachment(100,-5);
    data3.left = new FormAttachment(25,5);
    button3.setLayoutData(data3);
    . . .
}
```

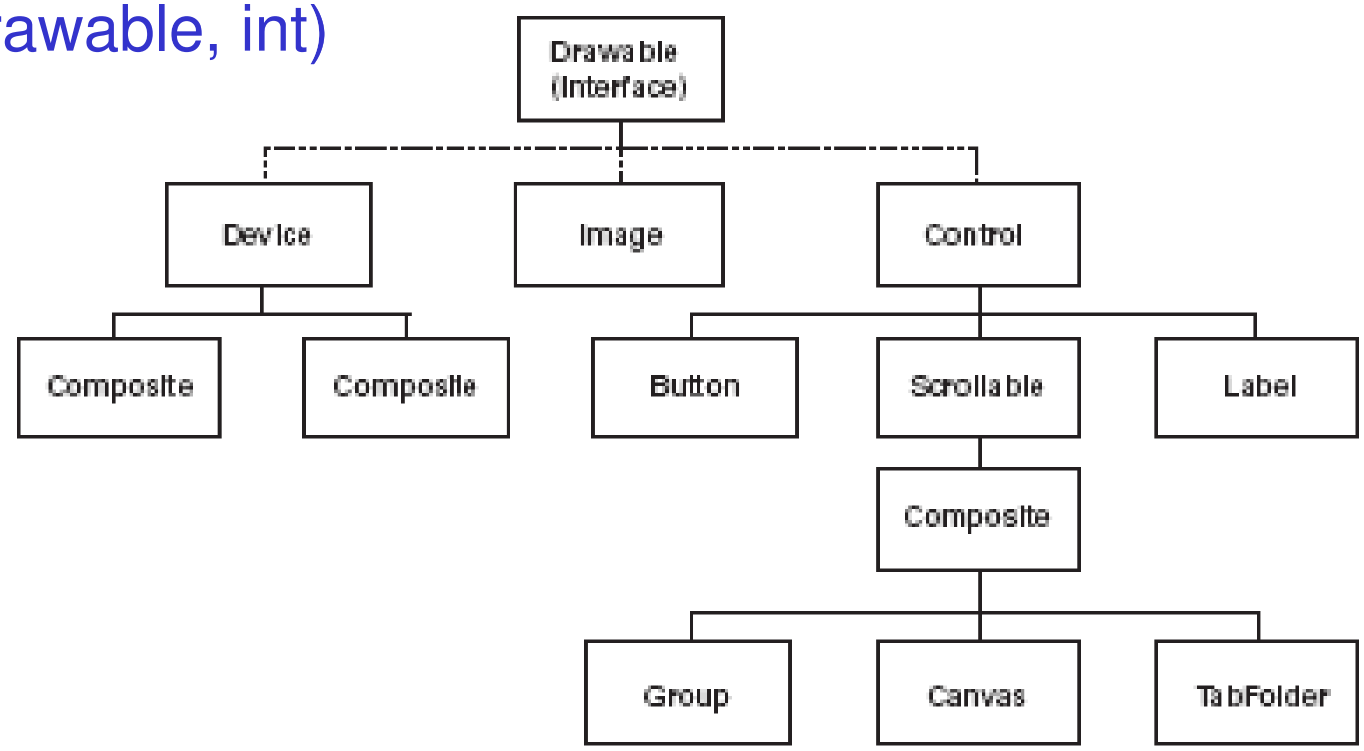
DEMO

## Graphics

- The graphic context functions like a drawing board on top of a Control
  - It lets you add custom shapes, images, and multifont text to GUI components
- It also provides event processing for these graphics by controlling when the Control's visuals are updated
- In SWT/JFace, the graphic context is encapsulated in the GC class
  - GC objects attach to existing Controls and make it possible to add graphics

## GC Class

- Creates a graphic and associate it with a component
  - GC(Drawable)
  - GC(Drawable, int)



## GC: Example

```
public class DrawExample {
    public static void main (String [] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Drawing Example");
        Canvas canvas = new Canvas(shell, SWT.NONE);
        canvas.setSize(150, 150);
        canvas.setLocation(20, 20);
        shell.open ();
        shell.setSize(200,220);
        GC gc = new GC(canvas);
        gc.drawRectangle(10, 10, 40, 45);
        gc.drawOval(65, 10, 30, 35);
        gc.drawLine(130, 10, 90, 80);
        gc.drawPolygon(new int[] {20, 70, 45, 90, 70, 70});
        gc.drawPolyline(new int[] {10,120,70,100,100,130,130,75});
        gc.dispose();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch())
            display.sleep();
        }
        display.dispose();
    }
}
```

DEMO

## Painting and PaintListeners

- When a GC method draws an image on a Drawable object, it performs the painting process only once
  - If a user resizes the object or covers it with another window, its graphics are erased
  - Therefore, it's important that an application maintain its appearance whenever its display is affected by an external event
- These external events are called **PaintEvents** and the interfaces that receive them are **PaintListeners**
  - A Control triggers a PaintEvent any time its appearance is changed by the application or through outside activity
  - Because a PaintListener has only one eventhandling method, no adapter class is necessary

## Painting Example

```
Canvas canvas = new Canvas(shell, SWT.NONE);
canvas.setSize(150, 150);
canvas.setLocation(20, 20);
canvas.addPaintListener(new PaintListener() {
    public void paintControl(PaintEvent pe) {
            GC gc = pe.gc;
            gc.drawPolyline(
            new int[]{10,120,70,100,100,130,130,75});
            }
    });
shell.open();
```

DEMO

## Painting Tips

- An interesting aspect of using PaintListeners is that each PaintEvent object contains its own GC
  - The GC instance is created by the event, the PaintEvent takes care of its disposal
  - The application can create the GC before the shell is opened, which means that graphics can be configured in a separate class
- Designers strongly recommend painting with Controls only in response to PaintEvents
- If an application must update its graphics for another reason, use the control's **redraw()** method
  - It adds a paint request to the queue
- Afterward, you can invoke the **update()** method to process all the paint requests associated with the object
- **NOTE**
  - **Although painting in a PaintListener is recommended for Control objects, Device and Image objects can't use this interface. If you need to create graphics in an image or device, you must create a separate GC object and dispose of it when you're finished.**

AOT LAB

**A**gent and **O**bject **T**echnology **Lab**
Dipartimento di Ingegneria dell’Informazione
Università degli Studi di Parma

# Standard Widget Toolkit and JFace

**Alessandro Negri**

negri@ce.unipr.it

http://www.ce.unipr.it/people/negri/