vogella.com

Eclipse SWT and JFace dialogs

Lars Vogel (c) 2010 - 2023 vogella GmbH

10–13 minutes

3.1. Dialogs from JFace

JFace contains several frequently used dialogs which are not based on the native dialogs as well as a framework for building custom dialogs.

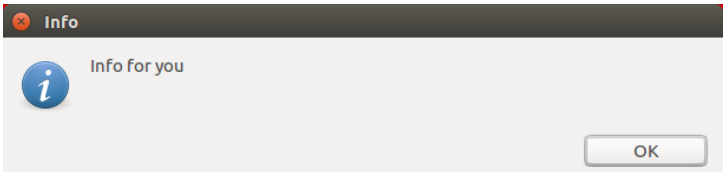
> Even though JFace dialogs are not native, they follow the native platform semantics for things like the button order.

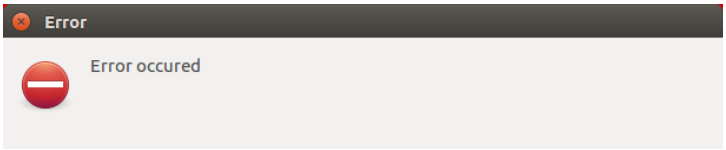
3.2. Using the static helper methods of the JFace MessageDialog class

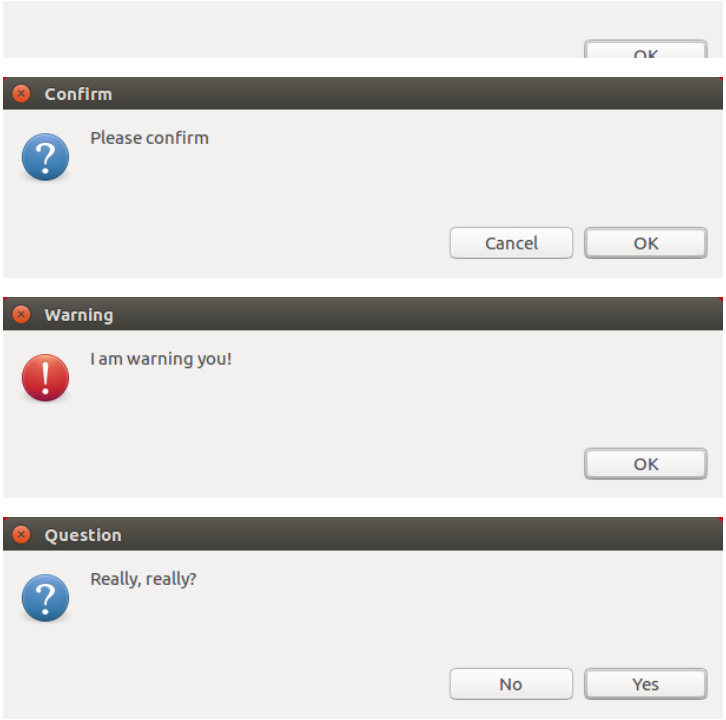
The `MessageDialog` class provides static methods to open commonly used dialogs, for example an information or a warning dialog. The following code demonstrates the usage of these static methods.

```
// standard message dialogs
MessageDialog.openConfirm(shell, "Confirm", "Please
confirm");
MessageDialog.openError(shell, "Error", "Error
occured");
MessageDialog.openInformation(shell, "Info", "Info for
you");
MessageDialog.openQuestion(shell, "Question", "Really,
really?");
MessageDialog.openWarning(shell, "Warning", "I am
warning you!");
```

The resulting dialogs are depicted in the following screenshots.





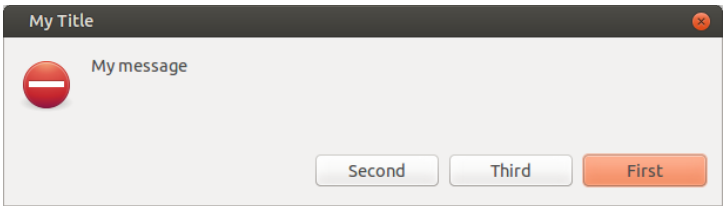


3.3. Using the JFace MessageDialog class directly

The `MessageDialog` class also allows the customization of the buttons in the dialog. The following code demonstrates its usage.

```
// customized MessageDialog with configured buttons
MessageDialog dialog = new MessageDialog(shell, "My
Title", null,
    "My message", MessageDialog.ERROR, new String[] {
"First",
    "Second", "Third" }, 0);
int result = dialog.open();
System.out.println(result);
```

If you open this dialog, it looks similar to the following screenshot.



Several of these dialogs return the user selection, e.g. the `openConfirm()` method returns true if the user selected the **OK** button. The following example code prompts the user for confirmation and handles the result.

```
boolean result =
    MessageDialog.openConfirm(shell, "Confirm",
"Please confirm");

if (result){
 // OK Button selected do something
```

```
} else {
 // Cancel Button selected do something
}
```

3.4. ErrorDialog

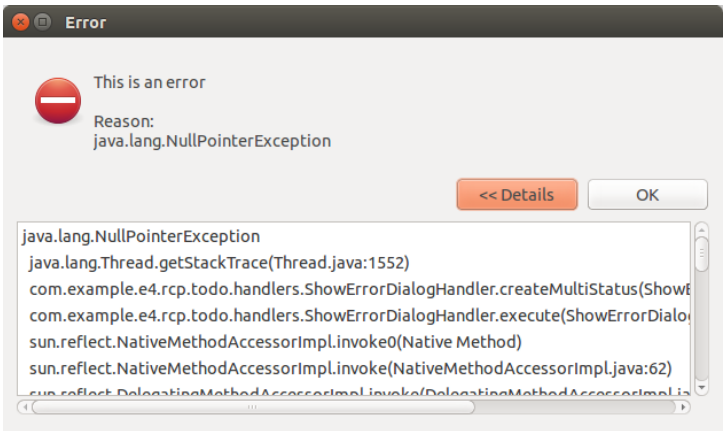
The `ErrorDialog` class can be used to display one or more errors to the user. If an error contains additional detailed information then a button is automatically added, which shows or hides this information when pressed by the user.

The following snippet shows a handler class which uses this dialog.

```
package com.vogella.tasks.ui.handlers;

import java.util.ArrayList;
import java.util.List;

import org.eclipse.core.runtime.IStatus;
import org.eclipse.core.runtime.MultiStatus;
import org.eclipse.core.runtime.Status;
import org.eclipse.e4.core.di.annotations.Execute;
import
org.eclipse.e4.ui.model.application.ui.basic.MWindow;
import org.eclipse.jface.dialogs.ErrorDialog;
import org.eclipse.swt.widgets.Shell;

public class ShowErrorDialogHandler {
    @Execute
    public void execute(final Shell shell, MWindow
window) {
        // create exception on purpose to demonstrate
ErrorDialog
        try {
            String s = null;
            System.out.println(s.length());
        } catch (NullPointerException e) {
            // build the error message and include the
current stack trace
            MultiStatus status =
createMultiStatus(e.getLocalizedMessage(), e);
            // show error dialog
            ErrorDialog.openError(shell, "Error",
"This is an error", status);
        }
    }

    private static MultiStatus
createMultiStatus(String msg, Throwable t) {

        List<Status> childStatuses = new
```

```
ArrayList<>();
        StackTraceElement[] stackTraces =
Thread.currentThread().getStackTrace();

        for (StackTraceElement stackTrace:
stackTraces) {
            Status status = new Status(IStatus.ERROR,
                    "com.vogella.tasks.ui",
stackTrace.toString());
            childStatuses.add(status);
        }

        MultiStatus ms = new
MultiStatus("com.vogella.tasks.ui",
                IStatus.ERROR,
childStatuses.toArray(new Status[] {}),
                t.toString(), t);
        return ms;
    }
}
```

If you trigger this handler, the dialog shows the exception messages and the detail page contains the stacktrace, as depicted in the following screenshot.



3.5. Creating a custom dialog

The `org.eclipse.jface.dialogs.Dialog` class can be extended to create your own dialog implementation. This class creates an area in which you can place controls and add an *OK* and **Cancel** button (or other custom buttons).

Your class needs to implement the `createDialogArea()` method. This method gets a `Composite` which expects to get a `GridData` object assigned as its layout data. Via the `super.createDialogArea(parent)` method call, you can create a `Composite` to which you can add your controls. This is demonstrated by the following example code.

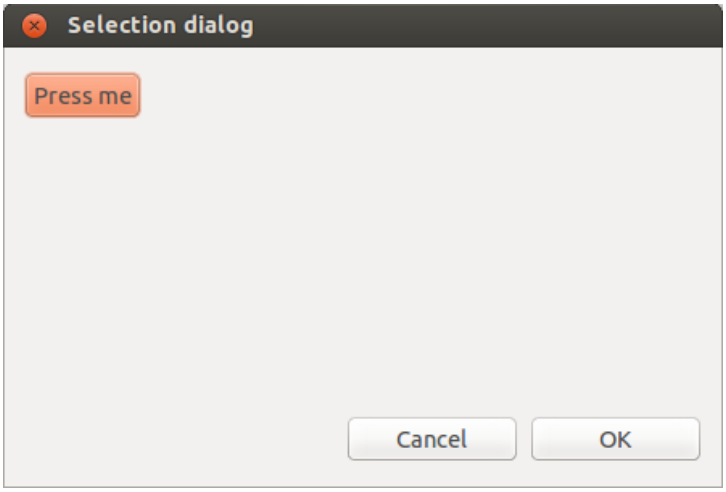
```
package com.vogella.plugin.dialogs.custom;
```

```
import org.eclipse.jface.dialogs.Dialog;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.graphics.Point;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;
import org.eclipse.swt.widgets.Shell;

public class MyDialog extends Dialog {

    public MyDialog(Shell parentShell) {
        super(parentShell);
    }

    @Override
    protected Control createDialogArea(Composite
parent) {
        Composite container = (Composite)
super.createDialogArea(parent);
        Button button = new Button(container,
SWT.PUSH);
        button.setLayoutData(new
GridData(SWT.BEGINNING, SWT.CENTER, false,
                false));
        button.setText("Press me");
        button.addSelectionListener(new
SelectionAdapter() {
            @Override
            public void widgetSelected(SelectionEvent
e) {
                System.out.println("Pressed");
            }
        });

        return container;
    }

    // overriding this methods allows you to set the
    // title of the custom dialog
    @Override
    protected void configureShell(Shell newShell) {
        super.configureShell(newShell);
        newShell.setText("Selection dialog");
    }

    @Override
```

```
    protected Point getInitialSize() {
        return new Point(450, 300);
    }

}
```

> The example code demonstrates how to set the title of your custom dialog via the `configureShell()` method.

If you open this dialog it looks similar to the following screenshot.



3.6. TitleAreaDialog

You can also implement your custom dialog based on the `TitleAreaDialog` class.

`TitleAreaDialog` has a reserved space for providing feedback to the user. You can set the text in this space via the `setMessage()` and `setErrorMessage()` methods.

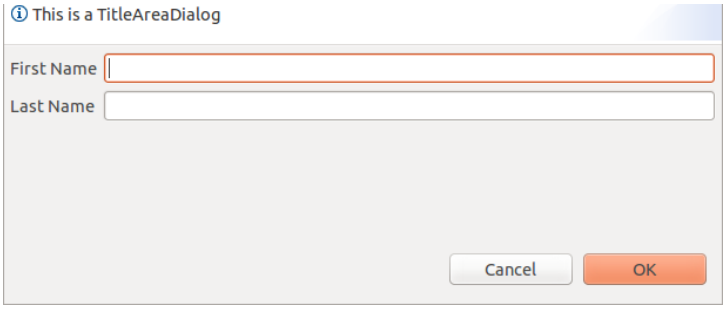
The following example shows a custom defined `TitleAreaDialog`.

```
package com.vogella.plugin.dialogs.custom;

import org.eclipse.jface.dialogs.IMessageProvider;
import org.eclipse.jface.dialogs.TitleAreaDialog;
import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Text;

public class MyTitleAreaDialog extends TitleAreaDialog
{

    private Text txtFirstName;
```

```
    private Text lastNameText;

    private String firstName;
    private String lastName;

    public MyTitleAreaDialog(Shell parentShell) {
        super(parentShell);
    }

    @Override
    public void create() {
        super.create();
        setTitle("This is my first custom dialog");
        setMessage("This is a TitleAreaDialog",
IMessageProvider.INFORMATION);
    }

    @Override
    protected Control createDialogArea(Composite
parent) {
        Composite area = (Composite)
super.createDialogArea(parent);
        Composite container = new Composite(area,
SWT.NONE);
        container.setLayoutData(new GridData(SWT.FILL,
SWT.FILL, true, true));
        GridLayout layout = new GridLayout(2, false);
        container.setLayout(layout);

        createFirstName(container);
        createLastName(container);

        return area;
    }

    private void createFirstName(Composite container)
{
        Label lbtFirstName = new Label(container,
SWT.NONE);
        lbtFirstName.setText("First Name");

        GridData dataFirstName = new GridData();
        dataFirstName.grabExcessHorizontalSpace =
true;
        dataFirstName.horizontalAlignment =
GridData.FILL;

        txtFirstName = new Text(container,
SWT.BORDER);
        txtFirstName.setLayoutData(dataFirstName);
```

```
    }

    private void createLastName(Composite container) {
        Label lbtLastName = new Label(container,
SWT.NONE);
        lbtLastName.setText("Last Name");

        GridData dataLastName = new GridData();
        dataLastName.grabExcessHorizontalSpace = true;
        dataLastName.horizontalAlignment =
GridData.FILL;
        lastNameText = new Text(container,
SWT.BORDER);
        lastNameText.setLayoutData(dataLastName);
    }



    @Override
    protected boolean isResizable() {
        return true;
    }

    // save content of the Text fields because they
get disposed
    // as soon as the Dialog closes
    private void saveInput() {
        firstName = txtFirstName.getText();
        lastName = lastNameText.getText();

    }

    @Override
    protected void okPressed() {
        saveInput();
        super.okPressed();
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }
}
```

This dialog is depicted in the following screenshot.



The usage of this dialog is demonstrated in the following code snippet. This code might for example be used in a handler.

```
MyTitleAreaDialog dialog = new
MyTitleAreaDialog(shell);
dialog.create();
if (dialog.open() == Window.OK) {
    System.out.println(dialog.getFirstName());
    System.out.println(dialog.getLastName());
}
```

3.7. Creating a non-modular dialog

You can use the `setShellStyle` method to create a non-modular dialog.

```
package com.vogella.tasks.ui.handlers;

import org.eclipse.jface.dialogs.Dialog;
import org.eclipse.swt.SWT;
import org.eclipse.swt.graphics.Point;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;
import org.eclipse.swt.widgets.Shell;

public class MyNonModularDialog extends Dialog {

    public MyNonModularDialog(Shell parentShell) {
        super(parentShell);
        setShellStyle(SWT.CLOSE | SWT.MODELESS |
SWT.BORDER | SWT.TITLE);
        setBlockOnOpen(false);
    }

    @Override
    protected Control createDialogArea(Composite
parent) {
        Composite container = (Composite)
super.createDialogArea(parent);
        Button button = new Button(container,
SWT.PUSH);
```

```
        button.setLayoutData(new
GridData(SWT.BEGINNING, SWT.CENTER, false, false));
        button.setText("Press");
        return container;
    }

    @Override
    protected Point getInitialSize() {
        return new Point(450, 300);
    }
}
```