Content assist

Content assist allows you to provide context sensitive content completion upon user request. This functionality is implemented by the platform text framework in **org.eclipse.jface.text.contentassist**. Popup windows (infopops) are used to propose possible text choices to complete a phrase. The user can select these choices for insertion in the text. Content assist also supports contextual infopops for providing the user with information that is related to the current position in the document.

Implementing content assist is optional. By default, **SourceViewerConfiguration** does not install a content assistant since it does not know the document model used for a particular editor, and has no generic behavior for content assist.

In order to implement content assist, your editor's source viewer configuration must be configured to define a content assistant. This is done in the Java editor example inside the **JavaSourceViewerConfiguration**.

```
public IContentAssistant getContentAssistant(ISourceViewer sourceViewer) {

        ContentAssistant assistant= new ContentAssistant();
        assistant.setContentAssistProcessor(new JavaCompletionProcessor(), IDocument.DEFAULT_CON
        assistant.setContentAssistProcessor(new JavaDocCompletionProcessor(), JavaPartitionScann

        ...
        return assistant;
}
```

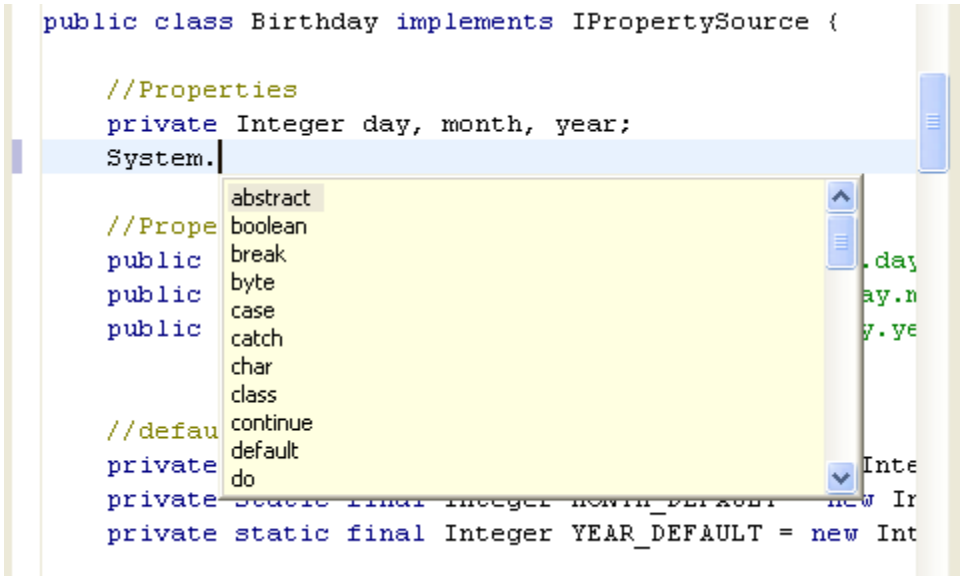
Content assist behavior is defined in the interface **IContentAssistant**. Setting up a content assistant is somewhat similar to setting up syntax highlighting. The assistant should be configured with different phrase completion strategies for different document content types. The completion strategies are implemented using **IContentAssistProcessor**. A processor proposes completions and computes context information for an offset within particular content type.

Content assist processors

Not all content types need to have content assistance. In the Java example editor, content assist processors are provided for the default content type and javadoc, but not for multi-line comments. Let's look at each of these processors.

The **JavaCompletionProcessor** is quite simple. It can only propose keywords as completion candidates. The keywords are defined in a field, `fgProposals`, and these keywords are always proposed as the candidates:

```
public ICompletionProposal[] computeCompletionProposals(ITextViewer viewer, int documentOffset)
        ICompletionProposal[] result= new ICompletionProposal[fgProposals.length];
        for (int i= 0; i < fgProposals.length; i++) {
                IContextInformation info= new ContextInformation(fgProposals[i], MessageFormat.f
                result[i]= new CompletionProposal(fgProposals[i], documentOffset, 0, fgProposals
        }
        return result;
}
```



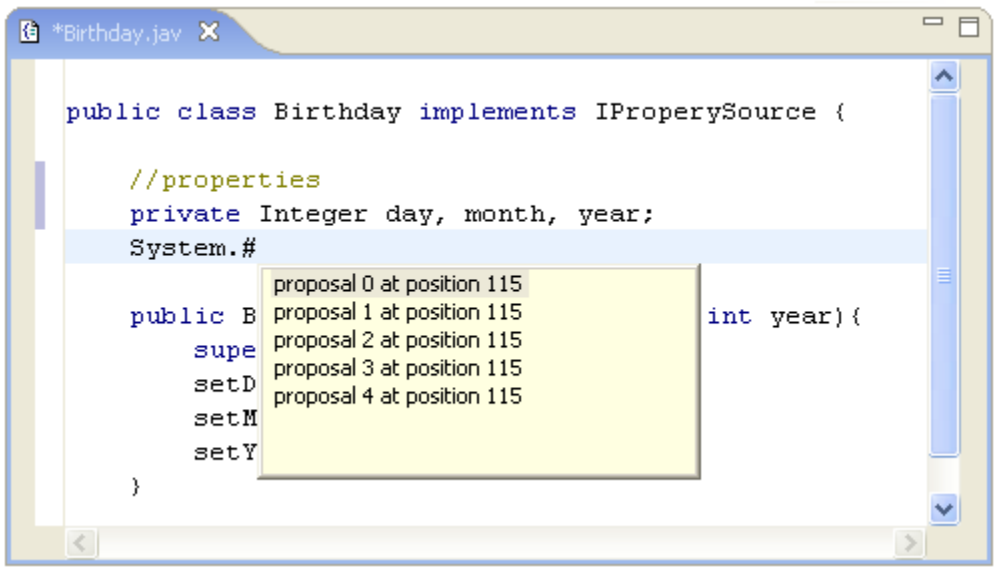
Completion can be triggered by user request or can be automatically triggered when the "(" or "." character is typed:

```
public char[] getCompletionProposalAutoActivationCharacters() {
        return new char[] { '.', '(' };
}
```

In addition to proposing completions, the **JavaCompletionProcessor** defines context information that can be requested by the user. Context information includes a description of the pieces of information available in a given context and the detailed information message.

In the Java editor example, the information is not really contextual. An array containing five similar context information objects is computed for the current offset when the user requests context info. All of these context information objects define a context that contains the five characters in front of the offset and the five after the offset. If any one of these five proposals is selected, the detailed information will appear near the cursor and will stay as long as the cursor is within the context of the five characters around the offset.

```
public IContextInformation[] computeContextInformation(ITextViewer viewer, int documentOffset) {
        IContextInformation[] result= new IContextInformation[5];
        for (int i= 0; i < result.length; i++)
                result[i]= new ContextInformation(
                        MessageFormat.format(JavaEditorMessages.getString("CompletionProcessor.C
                        MessageFormat.format(JavaEditorMessages.getString("CompletionProcessor.C
        return result;
}
```



This context information is shown automatically when the "#" character is typed:

```
public char[] getContextInformationAutoActivationCharacters() {
        return new char[] { '#' };
}
```

Content assist configuration

The appearance and behavior of content assist can be configured using **IContentAssistant**. For example, you can configure the auto activation time out, and the orientation and color of information popups.

```
public IContentAssistant getContentAssistant(ISourceViewer sourceViewer) {

        ContentAssistant assistant= new ContentAssistant();
        ...
        assistant.enableAutoActivation(true);
        assistant.setAutoActivationDelay(500);
        assistant.setProposalPopupOrientation(IContentAssistant.PROPOSAL_OVERLAY);
        assistant.setContextInformationPopupOrientation(IContentAssistant.CONTEXT_INFO_ABOVE);
        assistant.setContextInformationPopupBackground(JavaEditorEnvironment.getJavaColorProvide

        return assistant;
}
```

Create the content assist action and hook it with a key binding

In order to allow users to invoke content assist an action has to be created and configured. This is normally done in the sub-class implementation of `AbstractTextEditor.createActions()`:

```
protected void createActions() {
        ...
        IAction action= new ContentAssistAction(aResourceBundle, "ContentAssistProposal.", this)
        action.setActionDefinitionId(ITextEditorActionDefinitionIds.CONTENT_ASSIST_PROPOSALS);
        setAction(actionId, action); //$NON-NLS-1$
        markAsStateDependentAction(actionId, true); //$NON-NLS-1$
        PlatformUI.getWorkbench().getHelpSystem().setHelp(action, helpContextId);
        ...
}
```

The editor action bar contributor has to be extended to make the action appear in the main menu:

```
...
private RetargetTextEditorAction fContentAssist;

public MyEditorActionContributor() {
        fContentAssist= new RetargetTextEditorAction();
        String commandId= ITextEditorActionDefinitionIds.CONTENT_ASSIST_PROPOSALS;
        fContentAssist.setActionDefinitionId(commandId);
}

public void contributeToMenu(IMenuManager menu) {
        IMenuManager editMenu= menu.findMenuUsingPath(M_EDIT);
        editMenu.appendToGroup(MB_ADDITIONS, fContentAssist);
}

public void setActiveEditor(IEditorPart part) {
        IAction editorAction= getAction(part, "ContentAssist");
        fContentAssist.setAction(editorAction);
}
...
```