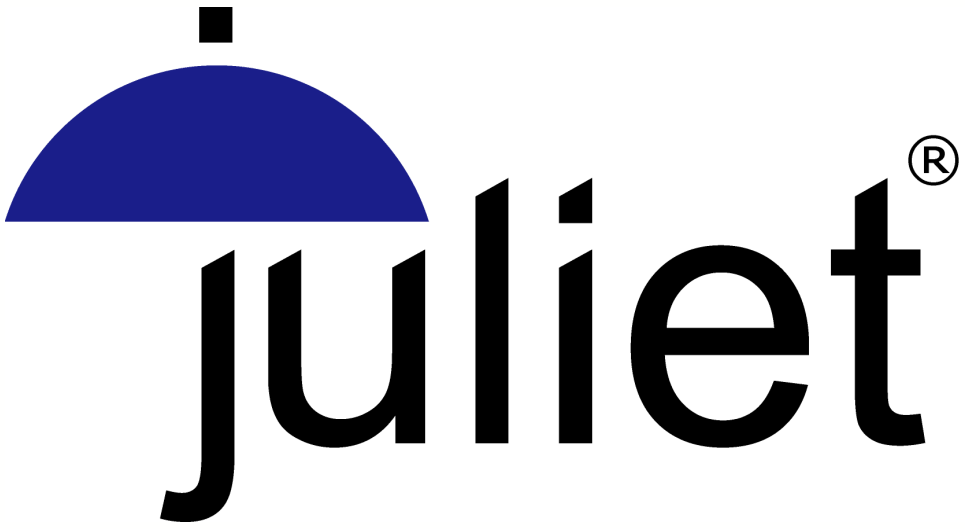


Juliet 2003a Documentation

Table of Contents

Tutorial / MUST READ!

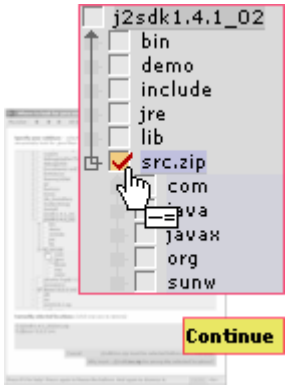
1 Tutorial / MUST READ!

1.1 Specify a codebase

To work through this tutorial (and on starting Juliet for the first time), you need to specify a **codebase** [one or more directories and/or archives (.zip, .jar, .war, .ear files) within which Juliet is to recursively look for source code]:

1. In any Juliet window, choose **MyJuliet** > **Manage your codebase** to jump to the *Manage your codebase* page. On that page, press the **Specify a new codebase** button to jump to the ... *Specify a codebase page*, that's right:

2. Use the file-system browser to select **src.zip** within your Java 2 SDK: it contains the source code - with embedded javadoc comments - for (most of) the standard Java libraries (***you need it to work through the example in this tutorial***):



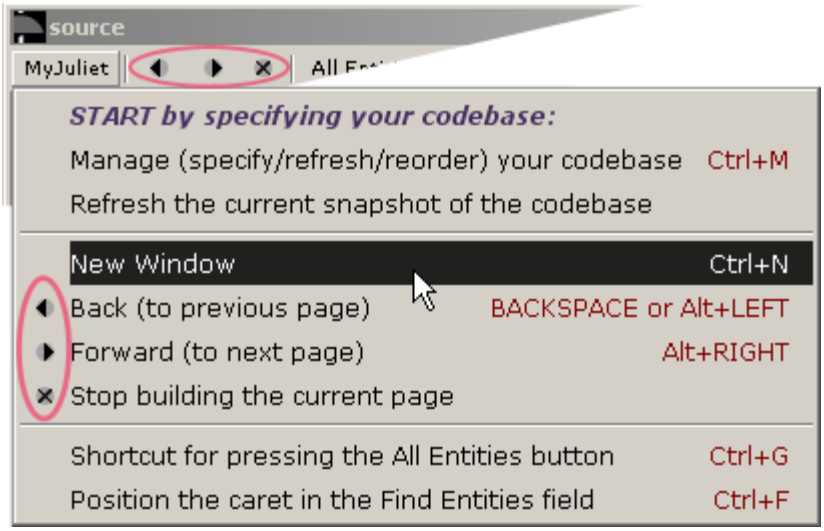
3. Press **Continue**: Juliet jumps back to the *Manage your codebase* page and starts building a snapshot. Think of a **snapshot** as Juliet's private on-disk database containing a copy of all the .java files in your codebase. Wait a bit until Juliet is done (note that it's quite fast, even though you just pointed it at half a million lines of source code).

4. After the snapshot is built, notice that on the lower right corner of each Juliet window a red progress bar appears and moves east to indicate how full Juliet's **cache** is. After a new snapshot is built, Juliet starts a background thread to build and cache a faster representation of some of the information in the snapshot: when the cache is full, some queries will run 50-100 times faster, but you can use Juliet normally while the cache is being filled (you only notice it because of the progress bar and because Juliet consumes quite a bit of CPU time until the cache is full).

1.2 Core concepts

Using Juliet is like using a web browser

All Juliet windows are the same and consist of a stable outer frame within which **pages** are displayed. **Press** the **MyJuliet** button to open the main popup menu:



Note the browser-familiar Back and Forward commands: each window maintains a history of previously visited pages, just like any browser window. Also note the **New Window** command: **you can have as many windows as you want**, **Ctrl+N** clones the current one - you will soon find this indispensable!

Every page is self-documenting via F1

Anytime and on any page, you can press the **F1** key to switch Juliet into "help mode": every visible area for which help is available will be outlined with a red rectangle. Move the mouse over any outlined area to see a detailed description about it (and click or press F1 again to move out of help mode and *freeze* the current help ballon - so that you can read it and scroll it) . **Note:** some of the active areas will be annotated with a numbered red bullet: if you want to understand what the current page is about, move the mouse over these bulleted areas in the given order. **Press** **F1 now** to try it, and remember to use it throughout this tutorial whenever you need more details!

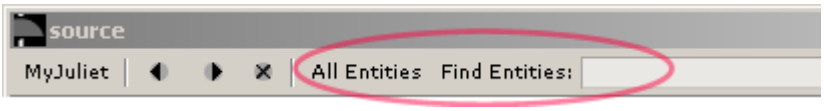
This manual (and especially this tutorial) does not explain every feature in Juliet - on purpose, because F1 does that for you, whenever you need it. Instead, this manual will explain some core concepts which would otherwise not be apparent, and give you a few usage examples.
press f1: juliet switches into help mode - ...
move the mouse around: each time it's over an active area, a balloon appears which contains an explanation of that area. some balloons are quite big, and the text in them needs to be scrolled for you to be able to read it AHA SIDE BOX: SCROLLING IN JULIET. the balloons are semi-transparent
press f1 again (or click the left mouse) when you want to freeze a balloon (it becomes opaque, and you can move the mouse into it, scroll it, etc) and switch juliet out of help moce
and again to get rid of the balloon completely

The fastest way to scroll is right-drag

You'll notice soon enough that Juliet doesn't use standard scrollbars, instead it displays 2 small triangles (one at each scrollbar extremity) which appear to be floating over the scrollable area. While you can still scroll the normal way by left-dragging over these triangles, it's much faster to simply **press the right mouse button** down anywhere **over the whole scrollable area** and drag the mouse up/down or left/right.

Every window has a "search bar" to navigate to *entities*

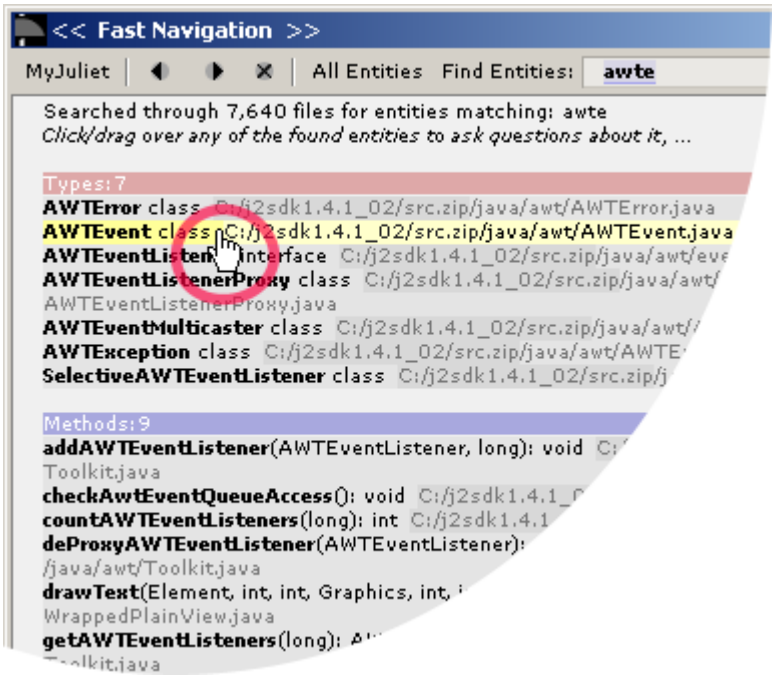
To the right of the browser (back, forward, stop) buttons you can see an **All Entities** button and a search bar called **Find Entities**:



Definition: an **entity** is one of:

.java file,
class,
interface,
constructor,
method,
field,
local variable or parameter.

The "Find Entities" search bar allows you to quickly locate any entity (except locals) and package as long as you can vaguely remember or guess its name. **Type** 'awte' in the search field (**Ctrl+F** positions the caret in it) - as soon as you enter 'a', Juliet jumps to a new *Fast Navigation* page which it updates while you type - and voilà, you've found java.awt.AWTEvent:



Now **move** the mouse over AWTEvent and note how the cursor changes to a HAND to indicate you're over a **Juliet link**. **Next**

1.3 Links and QueryPages

Juliet links

1. **Juliet links point to sets of one *or more* entities !**
2. you know you're over a link if the cursor changes to a HAND
3. you can **click**, **right-click** and **drag** over links.

The link you're over points to a set which contains the AWTEvent class.

Press (and keep pressed) **the left mouse button over the link to pick up the set of one or more entities the link points to** (*the mouse cursor changes to indicate that you have just picked up a set containing one entity, class AWTEvent*):

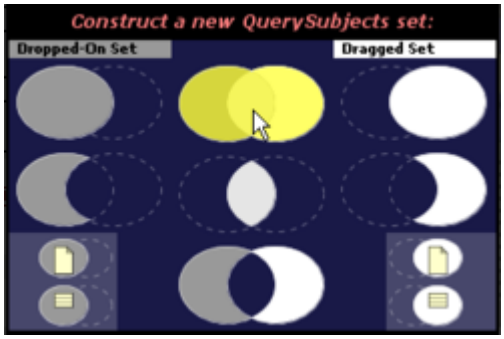
Release the left mouse button to drop the set over the current window:

Whenever you drop a set of entities over a window, Juliet reacts by jumping to a newly created QueryPage in the window over which you dropped the set:

The new QueryPage's *QuerySubjects* (**1**), *Query* (**2**) and *source area* are initialized as follows:

1. **QuerySubjects** (**1**): the dropped set is used as QuerySubjects, unless it was dropped over a QuerySubjects or QueryResults set, in which case you can combine both sets (union,

intersection, etc) to form a new set to be used as QuerySubjects:



2. **Query** (2): if the dropped set contains one entity, the "*Who references it?*" query is chosen, otherwise the simple "*Put all entities in QuerySubjects into QueryResults*" query is chosen.
3. **Choose what to display in the source area**: if QuerySubjects contains one entity, its declaration is shown, otherwise the source area stays empty (until you display something in it).

This explains why the just jumped-to QueryPage shows the declaration of AWTEvent in its source area, and answers ***Who references class AWTEvent?*** :

- its QuerySubjects set (1) contains the dropped AWTEvent class
- its Query (2) (about the single entity in QuerySubjects) is: "*Who references it?*"

So what's the **answer** to *Who references class AWTEvent* ***?*** **Next**

1.4 More on QueryPages

*The **answer** to a Query consists of a set of **annotated** entities (the **QueryResults** set). QueryResults can be **filtered** by **result category** to further **refine** the query (and/or by package, name, modifier or set-membership). **Result annotations** can (only) be seen and browsed in the source area.*

Answer == Annotated QueryResults (*which can be **refined***)

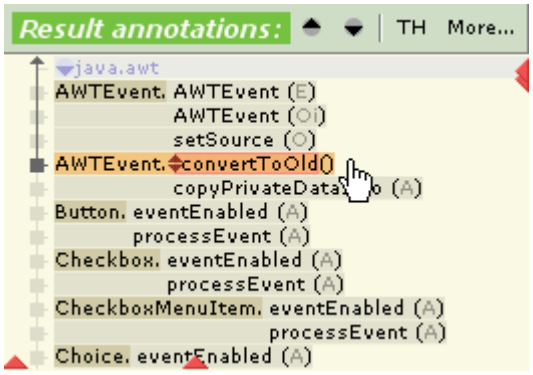
To use/understand a Query, you have to know:

1. which entities it puts into **QueryResults** (3)
2. how it **annotates** each entity in QueryResults: each entity in QueryResults has a **list** of annotations - each annotation in the list specifies a (sequence of) word(s) to be highlit in a source file.
3. into which **results categories** (5) it divides the results (you can ask more precise questions by limiting results to those of certain categories only)

If you were to press **F1** and move the mouse over the white-on-blue query label (**2**) you could see the definition of the "*Who references it?*" query, which would tell you that:

- **QueryResults** contains all entities which reference AWTEvent at least once (ie contains the answer to the question *Who uses AWTEvent?*);
- each entity in QueryResults has one **annotation** for each use of AWTEvent (ie contains the answer to the question *Where is AWTEvent used within this entity?*)

The results area (**4**) lists all entities in QueryResults, ordered alphabetically and by containing package. **Click** over the collapsed *java.awt* package header (the first line in the results area) to see all entities in QueryResults which are members of the java.awt package:



To look at **all of** convertToOld()'s annotations:

1. **Click** over **convertToOld()** to see its first annotation displayed in the source area: the first word within convertToOld which refers to AWTEvent is shown and highlit in orange in the source area. However, there are 4 more words within convertToOld which refer to AWTEvent --> 2.
2. **Press** **PgDown** (or the button) to display the next annotation on convertToOld's list of annotations. **Press** PgDown a total of 4 times and note how the various uses of AWTEvent

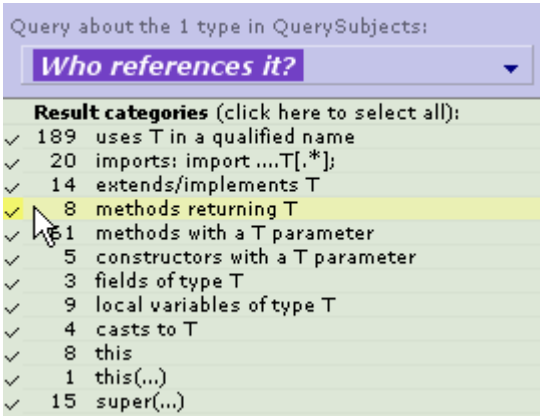
within convertToOld are highlit in the source area. If you pressed PgDown a 5th time, Juliet would show the first annotation of the next entity in QueryResults (the copyPrivateDataInto method).

Nota bene: when the mouse is over **convertToOld** it changes to a hand which means that you are over a link, yet when you click Juliet does **not** jump to a new QueryPage (*using convertToOld as QuerySubjects*) as you would expect. Why not? All links in this area, as well as all links in the area under QuerySubjects (which lists all entities in QuerySubjects) **react to clicks by changing what's displayed in the source area**. Instead of clicking try to left press and **drag** over **convertToOld**: the mouse cursor changes to a set and you have normal link behavior again, ie on **release** Juliet jumps to a new QueryPage. Try it, then use the Back button

Hint: most of the time, you simply use **PgDown** to cycle through all result annotations (of all entities in QueryResults). Simply repeatedly hit **PgDown** and look at the source area.

Result categories *(refining answers)*

Most queries subdivide their results into **result categories** which can be used to **refine** the query (*to ask a more precise question)* by filtering the results to those belonging to certain categories only. Depending on the query, they can also convey very useful information - for example a concise summary of ***how*** AWTEvent is used:



To **refine** the *Who references AWTEvent?* query to "***Which methods take AWTEvent as a parameter or return an AWTEvent*?**": **Click** over the metods returning T category *(the mouse is above it in the screenshot above)*, then **Ctrl Click** over methods with a T parameter to select both categories. Now repeatedly **press PgDown** to look at the result annotations (and note that the

QueryResults set now displays fewer entities).

Recursive (sub)queries

Position the mouse over the tree element to the left of **convertToOld** and note **1.** how the cursor changes to indicate that you can explode it and **2.** that a popup appears to the left which says **(Potentially) Called By**:

The popup indicates that on node expansion methods which (potentially) call **convertToOld** will be added as kids. **Click** over the node:

Subqueries allow you to recursively construct called-by, used-by and depends-on graphs, depending on which query is selected.

Changing the query

Click over the the white-on-blue query label: a popup menu appears which lists all available queries you can ask about the QuerySubjects. **Choose** **Supertypes & Subtypes**: Juliet jumps to a new QueryPage which uses the same QuerySubjects set as the previous page (ie class AWTEvent only), but whose QueryResults set contains all supertypes and subtypes, as well as the base type.

Concise type hierarchies

Now hit **Ctrl+H** or **press** over the **TH** button (it's to the left of the white-on-green ***Result annotations*** label): the area in which the query results are listed changes to a type hierarchy. Use **F1** to get more info on how this works. In a nutshell:

1.5 How to get to a QueryPage

1. On any QueryPage, **change the query** (ie ask another question about the QuerySubjects): press over the white-on-blue query label to open a popup menu which lists all currently available queries and choose one. Juliet jumps to a new QueryPage which uses the selected query and the same QuerySubjects as the current QueryPage.

2. In any Juliet window, **press** **All Entities** to jump to a QueryPage whose QuerySubjects set contains all entities (except for local variables) in your codebase. Normally, you only search through all files in your codebase for comments or string literals, which you can do using these shortcuts:

   Find comments: Ctrl+G (same as clicking over All Entities) followed by **Ctrl+Shift+C**. **Try it now**, try to find all comments containing both the words 'bug' and 'todo'!

Find string literals: Ctrl+G followed by **Ctrl+Shift+S**

Another frequent use for **All Entities** is to construct a QueryResults set which contains all entities declared **within a package and its subpackages**:

Click All Entities, [*choose the **Put them all into QueryResults** query, but it's selected by default since the QuerySubjects set contains several entities,*] then filter the QueryResults by package (use **F1** to find out how this works, hint: **java.awt+** would leave all entities declared in awt or a subpackage of awt in QueryResults).

3. Click over or drag and drop a Juliet **link** (remember, a link points to a set of one or more entities, which can be dropped over a window, or over a set area, for example over a QueryPage's QuerySubjects or QueryResults).

1.6 Using QueryResults as QuerySubjects

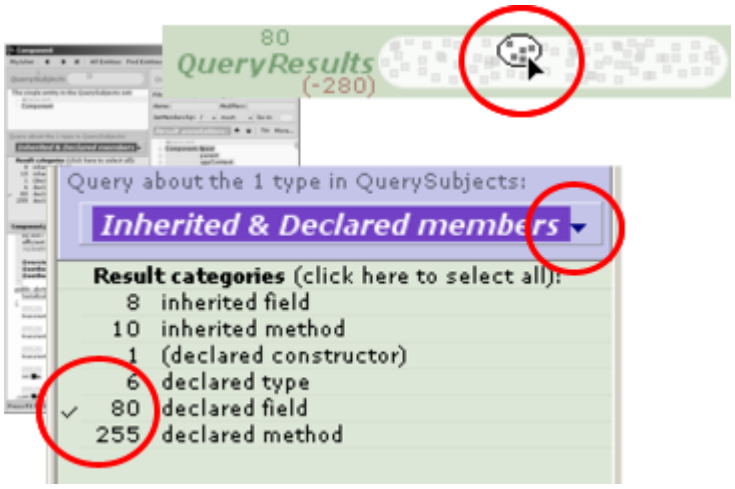
Let's work through an example:

Who writes to the fields of class java.awt.Component?

We need a QueryPage whose QuerySubjects contains all fields declared within java.awt.Component. Then simply choose the "Who references them?" query and refine it to "writes":

QuerySubjects = all fields declared in java.awt.Component
Query = "Who references them?", refined to "writes"

1. Find Component: choose any Juliet window, position the caret in its **Find Entities** field (**Ctrl+F**) and start typing "compo" until you can see java.awt.Component, then click over it to jump to a new QueryPage which uses Component as QuerySubjects.

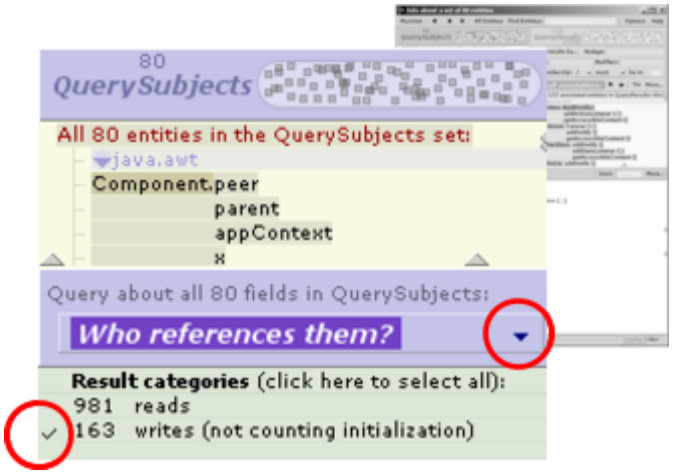


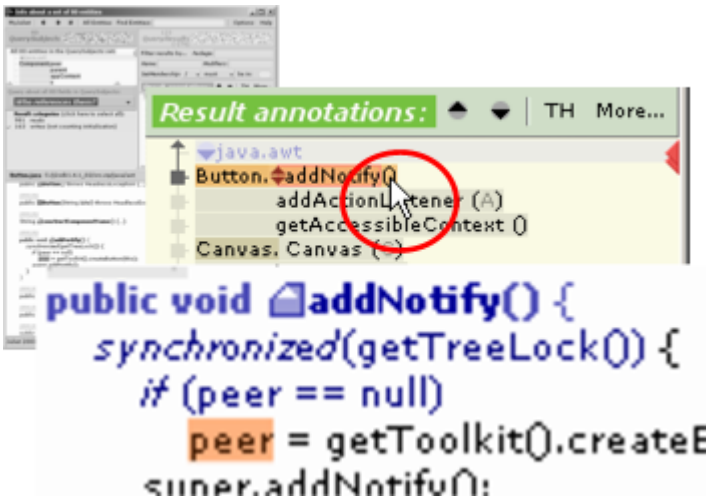
2. Click over the blue-on-white query label, choose the ***Inherited & Declared members*** query from the popup menu (this jumps to a new QueryPage) and...

3. ...refine it [click over the *declared field* result category] to only show ***declared fields***. the QueryResults set now contains the 80 fields declared in Component.

4. Click over QueryResults - or drag it over another Juliet window - to open a new QueryPage which uses the 80 fields as QuerySubjects.

Remember: QueryResults acts as a link, drag it over any window to get a QueryPage which uses it as QuerySubject!





5. Choose the ***Who references them?*** query and...

6. ...refine it to only show ***writes***: That's it: QueryResults now contains all entities within which one of the fields of Component is written to!

To see all places where fields in Component are written, you could now browse through all annotations by repeatedly hitting PgDown. Or, for example, click over the listed addNotify() method **to see its first annotation**: as you can see, the Component.peer field is updated within the addNotify() method.

To see if there are more writes to fields of Component within addNotify() (ie to see more than just the first annotation), hit PgDown or press the down arrow button next to the white-on-green "Result annotations:" label.

1.7 Using QueryResults to filter QueryResults

Another example to show how to answer:

> ***Does java.awt.Component or one of its subtypes [declare a method which] override[s] Object.finalize()?***

To answer this question you need to use the QueryResults from one query to filter the QueryResults of another query.

If *Component* itself declared a method overriding *Object.finalize()*, this would be easy: simply use that method as QuerySubjects (locate it via FindEntities and click over it) and choose the **Overriding and Overridden methods** query.

However, Component does *not* declare a method which overrides Object.finalize(). So instead, let's get

all methods overriding *Object.finalize()*, and then filter them to only those which are declared within direct or indirect subtypes of *Component*.

To do this, we need 2 Juliet windows:

- one window to query for all methods overriding Object.finalize(), and
- one window to query for all subtypes of Component.

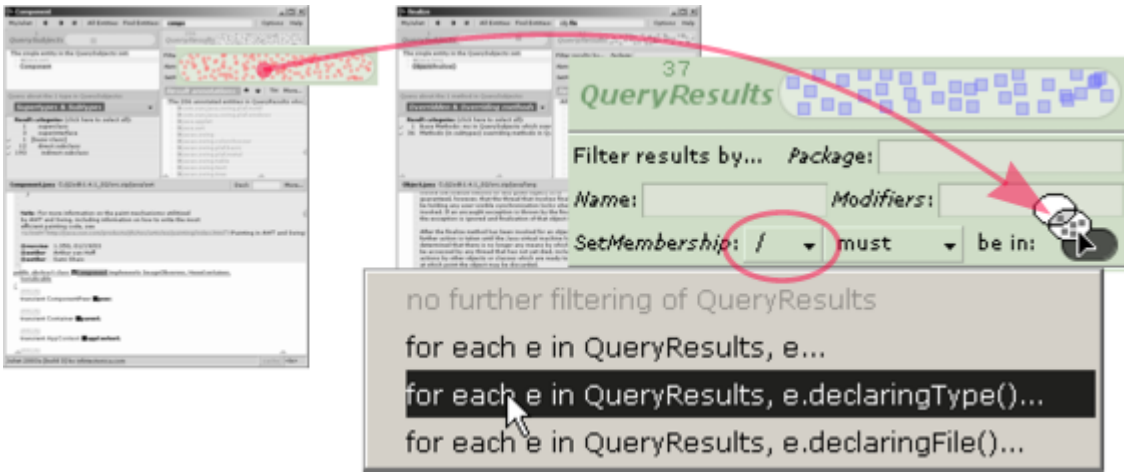
1. Get 2 windows: choose any Juliet window and clone it (use **Ctrl+N** or **MyJuliet** > **New window**). Arrange them next two each other (left and right) as shown above.

2. In the left window:

- Use **Find Entities** to jump to a QueryPage which uses Component as QuerySubjects [*Ctrl+F to position the caret in the Find Entities field, type "component" and click over the listed java.awt.Component*]
- Choose the **Supertypes & Subtypes** query [*click over the blue-on-white label and choose the query from the popup menu*] and...
- ...refine it to exclude supertypes (click over the ***base-class*** result category to limit query results to that category only, then **ctrl-click** over the ***direct subclass*** and ***indirect subclass*** categories to tick them as well): **QueryResults now contains Component and its direct and indirect subtypes**.

3. In the right window:

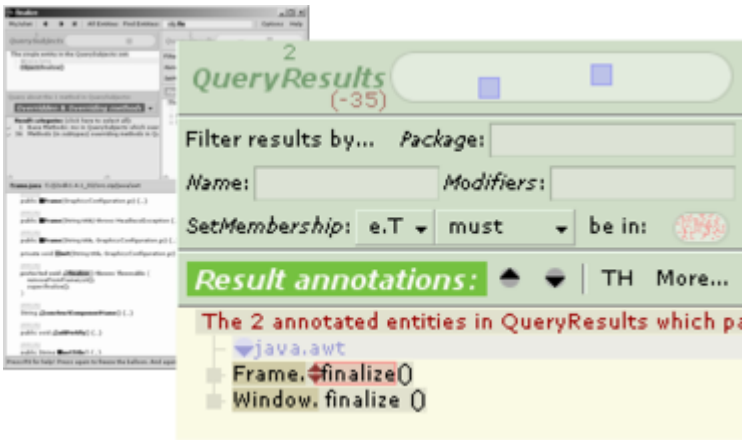
- Use **Find Entities** to jump to a QueryPage which uses Object.finalize() as QuerySubjects [*Ctrl+F to position the caret in the Find Entities field, type "obj.fin" and click over the listed java.lang.Object.finalize()*]
- Choose the **Overridden & Overriding methods** query [*click over the blue-on-white label and choose the query from the popup menu*]: **QueryResults now contains Object.finalize() and all methods that override it**.



4. Press the left mouse button over the QueryResults set in the **left** window (it contains Component and its subtypes), drag the set over the **FilterSet** in the **right** window and drop it.

5. Set up the filter so that only those methods in QueryResults whose declaring type is in the FilterSet (ie those methods which override finalize and are declared in a subtype of Component) pass.

That's it: the QueryResults set in the right window now contains the answer to the question "Do any of Component's subtypes override Object.finalize()": Only two types do: Frame and Window.



II

Available Queries

2 Available Queries

Note: more detailed info about queries (than the short description provided on the next 4 pages) is available directly from within Juliet: first select a query, then move the mouse over the white-on-blue query label and press F1 to see the exact query definition, which describes:

- the result categories;
- which entities are put into QueryResults;
- how these entities are annotated;
- (for queries which take extra input, such as searching for string literals) the format of the the input pattern.

Queries about all entities in QuerySubjects

Put them all into QueryResults
Who references them?
Whom do they reference?
Find code via JPattern

Queries about all files in QuerySubjects

Enclosed members
Find string literals
Find comments
Grep
Unresolvable imports

Queries about all types in QuerySubjects

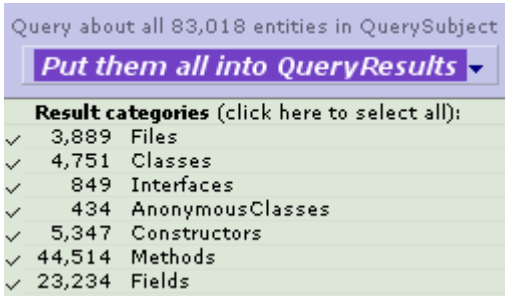
Enclosed members
Supertypes & Subtypes
Inherited and Declared members

Queries about all methods in QuerySubjects

Overridden & Overriding methods
Who (potentially) calls them?

2.1 Queries about all entities in QuerySubjects

Put all entities in QuerySubjects into QueryResults

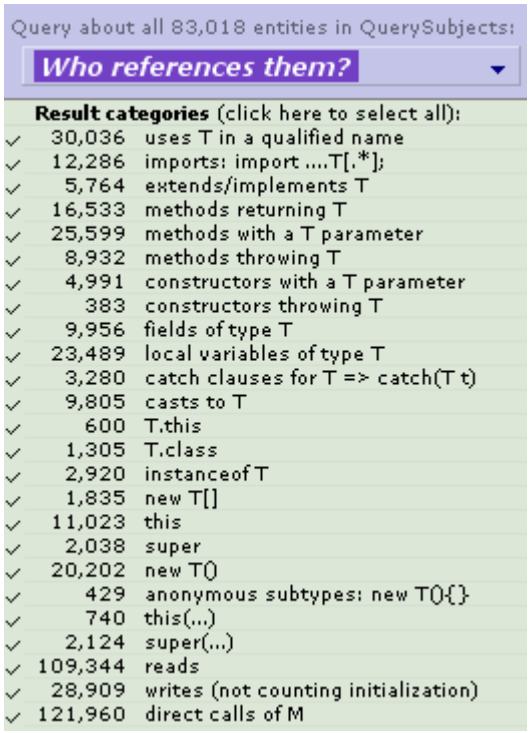


Simply puts each entity in QuerySubjects into QueryResults. Not very interesting as a query, but useful to construct subsets of QuerySubjects using one of the result filters.

Example use:

- To ask questions about all entities declared in a package *and* its subpackages: press the **All Entities** button, then use this query and apply the package filter (**java.awt+**). If you simply used the FindEntities to quickly locate and click over the awt package, you would only get the entities declared in awt, but not those declared in subpackages.

Who references the entities in QuerySubjects?



The most important and most often used query (and the query Juliet answers by default during semantic browsing / whenever a QuerySubjects set which contains only one entity is dropped over a window).

Answers 3 questions at once: *Who uses the entities in QuerySubjects? How are they used? Where are they used from?* "References" means: any using *identifier* or *this* or *super* token which resolves to one

of the entities in QuerySubjects.

The reason this query is so useful is because query results are categorized so that you can get answers to interesting questions such as: *Who instantiates the types in QuerySubjects? Who writes to the fields in QuerySubjects?*

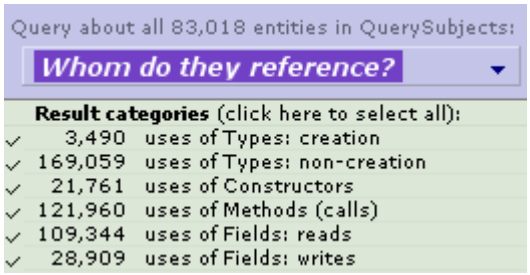
Example uses:

- *How is type T used?* Just by looking at the result categories you can already tell a lot (T might for example only be used in *instanceof* expressions, or never be directly created). You might also use type T *and* its declared members (see below for the query used to get them) as QuerySubjects to see how T's fields are used (maybe someone outside T wrties to them?), where its methods are called, etc.

- *Where is method M called from? Who writes to field F?* Etc, these simple questions are answered automatically while you browse.

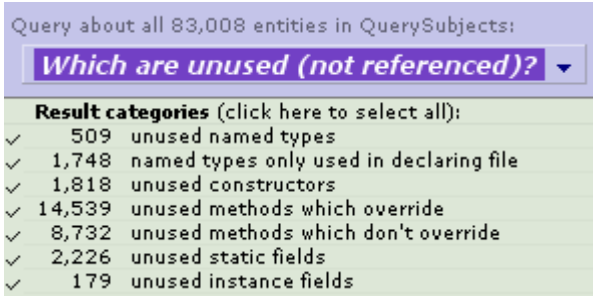
- *How is package p1.p2 used?* Use **Find Entities** to quickly locate and click over p2: this jumps you to a QueryPage using all types, methods, fields etc declared in p2 as QuerySubjects. Then select this query. If you also want to know how entities in p2's subpackages are used, you need to construct the QuerySubjects set via by using the **All Entities** button and applying the **p1.p2+** package filter.

Whom do the entities in QuerySubjects reference?



Answers the question: *Which entities are used by the entities in QuerySubjects?* The query results are categorized so that you can get answers to more specific questions such as: *Which fields do the entities within QuerySubjects write to? Which types are created by the entities in QuerySubjects, and where from?*

Which of the entities in QuerySubjects are unused (not referenced)?



Ever wondered if you had unused instance fields? This query returns all entities in QuerySubjects which are not referenced from anywhere within your codebase. Note that this is not the same as an "unreachable code" query, because if for example method f() calls method g(), but method f itself is never called, g is unreachable, but not shown as unused by this query because it is referenced from within f.

Find code via JPattern...

JPattern stands for "simple pattern matching over tokenized source code": it knows to ignore whitespace and comments to answer questions which are impossible to ask using tools like grep, such as: *Does our production code contain System.out.println() statements within catch blocks?*

For each entitiy in QuerySubjects, JPattern reads through its associated code segment (*for example for each method in QuerySubjects, JPattern scans through every token in that methods declaration, including the tokens in the method's body*) and tries to find matches for the pattern. You can for example put all methods overriding Object.clone() into QuerySubjects and then use JPattern to find which ones contain *super.* !

Some useful patterns:

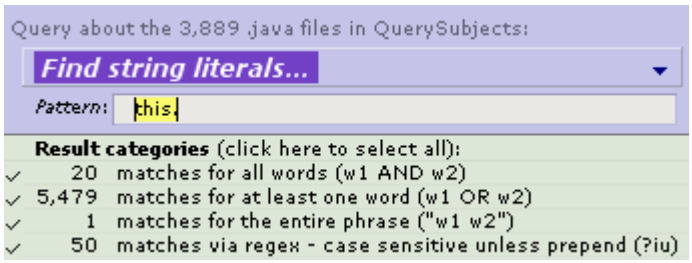
`catch(...){}` Finds empty catch blocks
`catch(...){ ...System.out.println...}` Finds catch blocks within which `println is called`

2.2 Queries about all files in QuerySubjects

Enclosed members

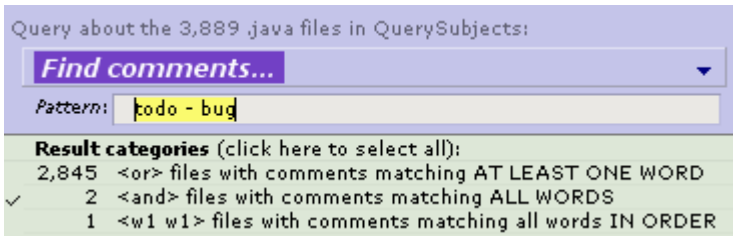
For each file in QuerySubjects, add to QueryResults each declaration that it contains *except for local variables and parameters*.

Find string literals...



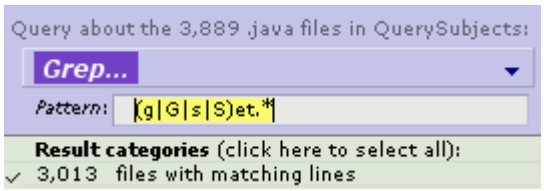
Used to quickly find string literals within the files in QuerySubjects. Juliet tries different matching strategies in parallel (*you can refine the results to exclude unwanted strings*): look for literal matches, look for strings containing all the words in the pattern, or interpret the pattern as a regular expression. Most often, you do a global search for string literals (it's fast because it uses an index) by first pressing the All Entities button to get a QueryPage whose QuerySubjects contains all the files in your codebase.

Find comments...



Used to quickly find comments within the files in QuerySubjects. Juliet splits the pattern into words and categorizes the results into: comments which contain all of the given words, comments which contain at least one of them, and comments which contain all of them in the given order. It's a fast query because it's indexed, and it's the fastest way we know of to search through javadoc comments. As for "*Find string literals*...", this is most often used with the All Entities button.

Grep



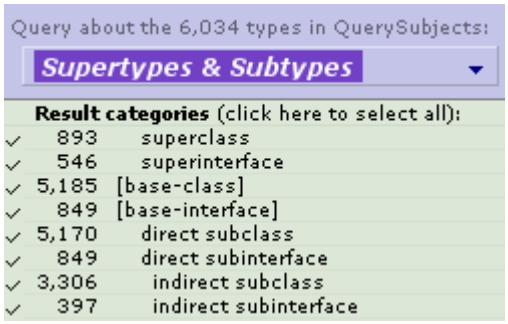
Only included to show that you don't need it any more :) Searches through all .java files in QuerySubjects line per line and tries to match each line with the input pattern (which is interpreted as a standard grep regular expression).

Unresolvable imports

Searches through all .java files in QuerySubjects and shows all import declarations which cannot be fully resolved: this indicates a codebase that's not complete. This is not a problem as long as (ideally) all of the code you actually are interested in is resolvable, ie as long as all of these imports are located within code that's used by the code you're interested in, but is not part of it.

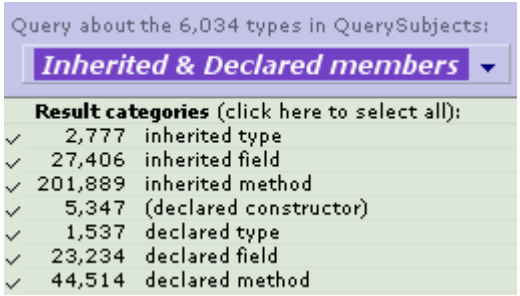
2.3 Queries about all types in QuerySubjects

Supertypes & Subtypes



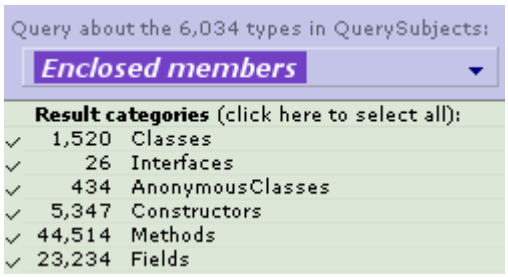
Puts each type in QuerySubjects into QueryResults, together with its direct and indirect super- and sub-types. You can refine the results so that you only have direct or indirect sub-types. This query is most often used together with Ctrl+H to see the results visualized as a type hierarchy (Ctrl+H).

Inherited & Declared members



Shows all declared and inherited members of all types in QuerySubjects. Most often used to construct a set containing a type T plus all of its declared members to use with a "*Who references*" query.

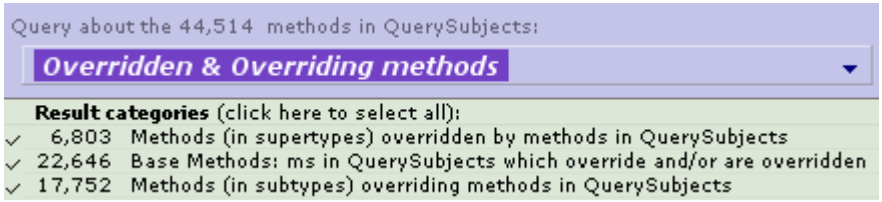
Enclosed members



For each type in QuerySubjects, scan through its body (*recursively looking into all enclosed blocks*) and add to QueryResults each declaration that is found *except for local variables and parameters*. For example: to find where all of the methods and fields within java.awt.Component and its nested types are referenced from, put Component into QuerySubjects, use this query to get its enclosed members, use them as new QuerySubjects and ask "*Who references them?*".

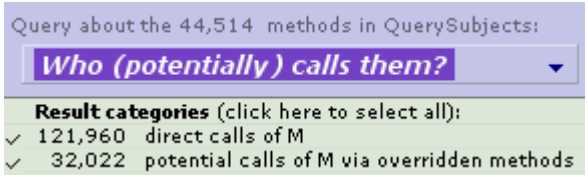
2.4 Queries about all methods in QuerySubjects

Who overrides / is overridden by the methods in QuerySubjects?

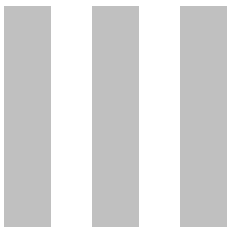


Shows, for each method M in QuerySubjects, which methods are overridden by M and which methods override M. Visualize the results using Ctrl+H to see which types declare overriding/overridden methods.

Who (potentially) calls the methods in QuerySubjects?



If you use the "*Who references them?*" query on methods, you only see direct calls. However, a method T2.m() might for example never be called directly, but that doesn't mean it's never invoked at runtime: if class T1 declared a method m() which is overridden by T2.m(), then if T1.m() is directly called via a type which is a supertype of T2 this could *potentially* result in a runtime call to T2.m(). This query shows both direct and potential calls to all the methods in QuerySubjects.



Manage your codebase

3 Manage your codebase

3.1 Overview

A **codebase** is a list of locations which tells Juliet where to look for source code (.java files). A **snapshot** is Juliet's private copy of all the .java files recursively located under the codebase locations. Whenever Juliet builds or updates a snapshot, it automatically deduces the **sourcepath**.

From the ***Manage your codebase*** page you can:

- **specify a new codebase**
- **update the current snapshot**
- **reorder the deduced sourcepath**

You can get to this page via **MyJuliet** > **Manage your codebase** (**Ctrl+M**) as well as via the **Continue** button on the *Specify your codebase* page.

The ***Specify your codebase page*** lets you switch between different (previously used) codebases or specify new ones (details).

You get to this page by pressing the **Specify a new codebase** button on the *Manage your codebase page*.

3.2 Codebase

A **codebase** is a list of locations within which Juliet is to recursively look for source code (.java files), for example:

```
c:/jdk/src.zip
```

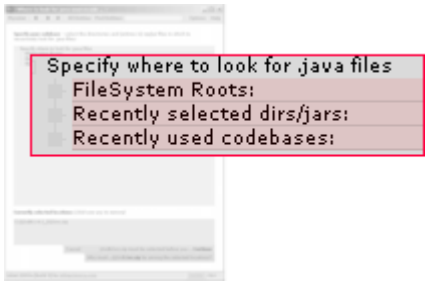
```
c:/projects/new
c:/libs/oldlib.jar/src
```

*A **location** is a directory, an archive (zip/jar/war/ear file), or a directory within an archive.*

How to specify a codebase

1. In any Juliet window, jump to the *Manage your codebase* page (**Ctrl+M** or **MyJuliet** > **Manage your codebase**), then press the **Specify a new codebase** button to jump to the *Specify your codebase* page on which you can select previously used codebases or specify new ones:

2. The *Specify your codebase* page gives you three options to select your codebase locations:



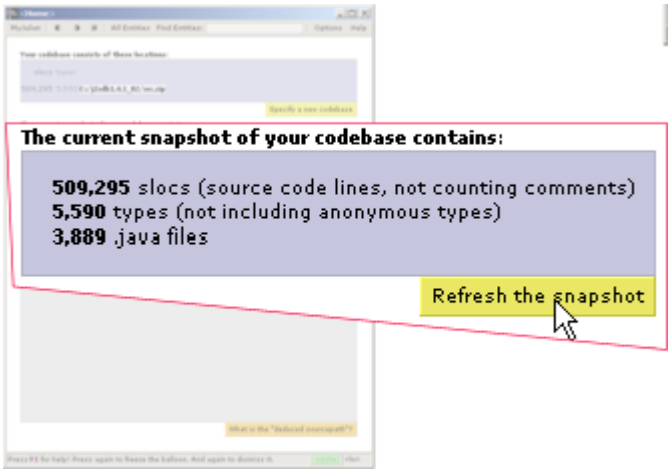
   - Expand **FileSystemRoots** and use Juliet's in-built browser to find and select locations. (*Note that you can browse into archives as if they were normal directories, and that only archives and directories which may contain .java files or archives are shown*).
   - Expand **Recently selected...** this works the same way as above but provides direct access to recently selected locations.
   - Expand **Recently used codebases** to quickly switch between previously specified codebases: if this is not your first use of Juliet, you will see a list of italian/american names representing previously selected codebases - Juliet automatically remembers the locations making up previously selected codebases and assigns unique names to them. You can expand each name to see which locations make up its codebase, or simply select its tickbox to (re)select all of its locations - this allows you to quickly switch between different codebases, we call it "***automatic project management***".

3. #ConfirmCodebaseSpec Once you have selected all locations you wish to be part of your current codebase, press **Continue**: Juliet jumps back to the *Manage your codebase* page and starts building a snapshot (while the snapshot gets built, Juliets also deduces the sourcepath).

3.3 Snapshot (of the codebase)

Juliet does not directly work from the files in the codebase but from a snapshot of these files. A **snapshot** is Juliet's private copy of all the .java files directly or indirectly located within the codebase locations (*stored on disk in one or more .xcpm files under Juliet's installation directory*).

Snapshots ensure that Juliet is isolated from changes to files in the codebase after the snapshot was taken - until you request a snapshot update:



A snapshot is taken (or updated: updating a snapshot is equivalent to taking a new one, but faster) whenever you:

- **SPECIFY** a codebase and press the **Continue** button to start the snapshot build process (here).
- **REFRESH** the snapshot, either by choosing **MyJuliet** > **Refresh the current snapshot of the codebase** or by pressing the **Refresh the snapshot button** on the *Manage your codebase* page.
- **START** Juliet. On start-up, the stored snapshot of the last used codebase is automatically loaded and updated - if you want to use a different codebase, interrupt the snapshot build process by pressing the **Stop** button (✖) and specify/select another codebase.

3.4 Deduced Sourcepath

A **sourcepath** is an ordered list of locations used to find the source code for types. For example if you told a java compiler (*javac*, *jikes, etc*) to use this sourcepath

```
c/project/src_new
c/project/src_old
```

it would try to find the source for class **`java.lang.Object`** as follows:

1. if **`c/project/src_new`**`/java/lang/Object.java` exists, use it
2. else if **`c/project/src_old`**`/java/lang/Object.java` exists, use it
3. else fail: can't find the source for java.lang.Object

If both src_new and src_old contained a `/lang/Object.java` file, the one in src_new would win because it comes first on the sourcepath. The order of the locations on the sourcepath matters, but

only if different locations contain the same type, because the first location in which the type is found **hides** the other declarations.

If instead you had told the compiler to use this sourcepath

```
c/project
```

it wouldn't have found Object because it would only have looked for
`c/project/``java/lang/Object.java` wich does not exist!

Unlike Juliet! If you tell Juliet to use `c/project` as your codebase, Juliet correctly deduces this sourcepath:

```
c/project/src_new
c/project/src_old
```

All you have to do [where necessary] is (re)order the deduced locations by dragging them into the wanted position. On this screenshot, the mouse is over **.../connector/src/main** which can now be dragged up/down in the lookup order:

So what? Look at the above screenshot: the codebase consists of two locations only (*src.zip* and *jboss-3.2.1-src*), yet the deduced sourcepath counts 29 locations (*not all visible above*). Juliet's automatic sourcepath deduction means you don't have to know the details of how a codebase is structured (very helpful when looking at someone else's code), and even when you do know the details it is convenient to have Juliet do the work.

3.5 What the codebase should contain!

To make best use of Juliet's semantic understanding of Java, your codebase should be as self-

contained as possible: ideally the source code for every class and interface which is used anywhere within the codebase should be in the codebase. **[Note: support for .class files is nearly ready and will be released "*in 2003, well before Christmas*".]**

To ensure that (most of) the code in your codebase can be understood by Juliet, make sure your codebase contains 1. the source code you are interested in browsing *plus* 2. the source code of all types used by your code! 99.999% of the time this means that your codebase should contain:

- **the src.zip file** shipped with the JDK download by sun (src.zip contains the source code for nearly all standard Java libraries), for example: C:/jdk/src.zip;
- **your source code**, for example: C:/goodJavaProjects and C:/uglyJavaProjects;
- **the source for all other libraries** directly used by your code (if any)

Explanation

What if - for example - you specified your codebase to consist of only the C:/myJavaProjects location? Juliet would very probably not be able to resolve all identifiers within your code! Consider this source file:

```
// Assume this .java file is stored in C:/myJavaProjects/myp1/myp2
package myp1.myp2;
public class A
{
  public void foo(String s)
  {
  }
  public void foo(A a)
  {
  }
  public static void main(String[] args)
  {
    A a = new A();
    String s = null;
    foo(s);
  }
}
```

Unless there is a .java file which declares a class or interface called *String* somewhere in C:/myJavaProjects, Juliet won't be able to resolve the identifiers shown in underlined red: your codebase is not complete, it doesn't contain source code for all used types!

Note that the foo(s) call in main also cannot be resolved: the *s* argument can be resolved to the local variable *s*, but the type of that local variable is unknown. Since method lookup requires both the method name and the parameter types to distinguish between overloaded methods (methods with the same name, but different parameter types), method lookup is undefined if any of the parameter types is unknown.

IV

Integrating Juliet with your IDE/Editor

4 Integrating Juliet with your IDE/Editor

4.1 How to launch your editor from Juliet

If a .java file is shown in the Juliet window with the input focus *and* if that file is not in an archive (`.../src.zip/java/lang/Object.java`) but in a directory, **Ctrl+E** opens that file in your favorite editor.

The first time you use **Ctrl+E**, Juliet asks you to specify which editor to use. You can (re)specify this any time: choose **Options** > **Choose your editor** and select the application which Juliet should invoke whenever you use **Ctrl+E**.

4.2 How to control Juliet from other applications

Depending on how you installed Juliet, you start Juliet via one of the following:

- a native launcher, for example: `Juliet_2003a.exe`
- a script: `Juliet.bat, juliet.sh, Juliet_2003a.bin`

For the discussion below, **juliet** designates the executable or script used to launch Juliet.

To control an already running Juliet instance from another application (for example to configure your favourite text editor so that each time you hit the ESCAPE key it causes Juliet to show information relevant to your cursor position in the editor), you have to configure that application to call **juliet** with extra arguments.

Whenever **juliet** is invoked, it checks if another instance of Juliet is already running, and if so it passes the extra arguments to that instance and exits. The already running instance looks at the arguments passed to it and reacts accordingly:

juliet -findEntity

Brings a Juliet window forward and positions the cursor in the FindEntity field (as if you had used Ctrl&F from within Juliet), ready to accept input. Use this command to augment your editor with Juliet's quick search capabilities.

juliet -typeInfo *typename*

typename
a fully qualified typename, for example: java.lang.Object

If a type with fully qualified name typename is found in Juliet's snapshot, Juliet will display information about that type (source code, used by, etc).

Example: `juliet -typeInfo java.lang.Thread`

juliet -contextInfo *filename linenum column*

filename
The absolute filepath of a .java file, for example:
`c:/projects/Project1/package1/package2/SomeFile.java`
linenum
Linenum and Column describe a caret position in the named file. Lines are counted starting with 1 (not 0) for the first line in a file.
column
Linenum and Column describe a caret position in the named file. Columns are counted starting with 1 (not 0) for the caret at the leftmost position in a line.

This command is meant to be used from within your editor, to request information relevant to the current caret position (passed in via *linenum* and *column*):

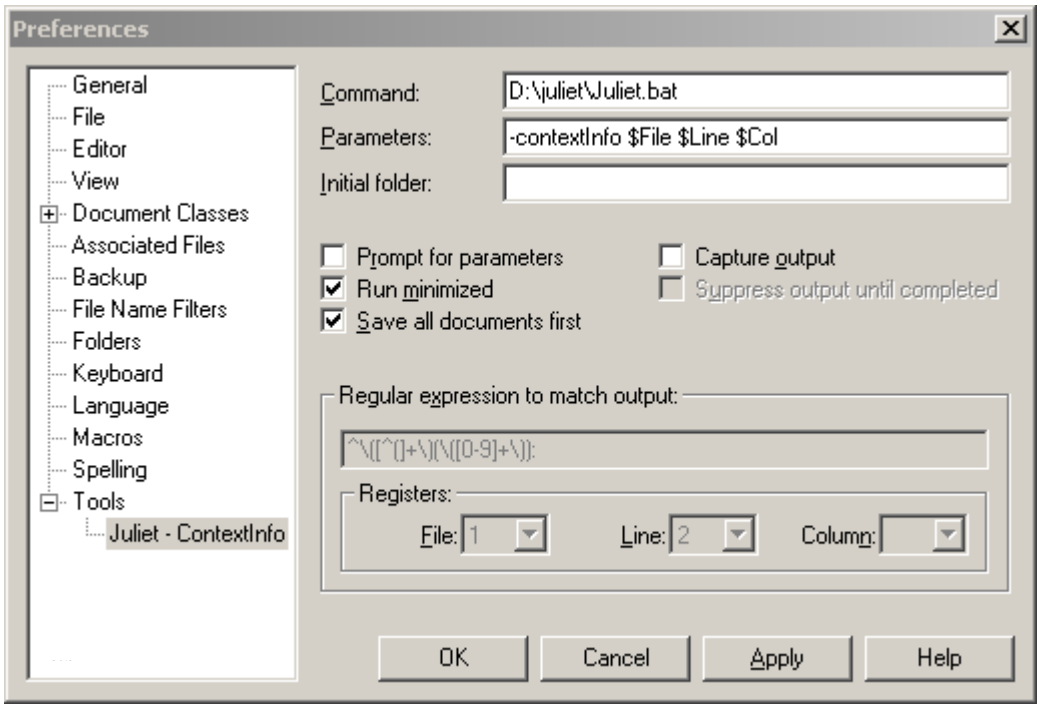
- If the caret is over a token (identifier or super or this) which refers to a declared entity which is not a local variable, Juliet displays information about that entity. To be precise: Juliet jumps to a Queries page with the QuerySubjects set initialized to contain the declared entity, and the selected query set to "Who references it?".
- If the caret is over a declaring identifier (of a class, interface, constructor, method or field), Juliet will display information about that declaration.
- Otherwise, Juliet jumps to the FindEntites page LINK and displays all search results for the word under the token (as if you had entered that word into the Find Entities: text box in Juliet).

Note: Juliet needs to figure out what the caret position means. To do this reliably, the *filename* file must be saved (by your editor) before invoking this command: the caret position is interpreted with respect to the saved file, not with respect to your editor's current buffer, because Juliet does not have access to your .
[Also (to be properly explained): this command causes Juliet to update its snapshot if it hasn't seen filename yet or if filename has been modified. Note that filename must be within one of the directories you told Juliet to monitor for source code, otherwise this command will be ignored.]

Example 1: `juliet -contextInfo "c:/src/java/lang/Object.java"  51 14`

Example 2: To integrate with jEdit, you can usethis beanshell macro (you need to edit one variable, read the comments in the macro).

Example 3: This is how you would configure TextPad (a Windows editor) to ask Juliet for context information while editing. It is recommended that you tell TextPad to display context information whenever Ctrl&Space or Escape is pressed (fast and convenient):



Any rants, raves, suggestions etc which would help to improve this manual are very welcome:

http://infotectonica.com/juliet/contact/