

## Debugging with Chronon

- [What this tutorial is about](#)
- [What this tutorial is not about](#)
- [Before you start...](#)
- [Preparing an example](#)
- [Installing plugin](#)
- [Changes to the UI](#)
- [Creating run/debug configuration](#)
- [Defining include/exclude patterns](#)
- [Running with Chronon](#)
  - [Opening an existing record](#)
- [What can you do with a record?](#)
  - [Switch between threads](#)
  - [Step through the application](#)
  - [Use bookmarks](#)
  - [Explore methods](#)
  - [Log values](#)
  - [Explore exceptions](#)
- [Summary](#)

### What this tutorial is about

This tutorial aims to walk you step-by-step through debugging a Java application with Chronon, a recorder and a "time-travelling" debugger. Chronon records changes made by your application while it is executing. The recordings are saved to files. You can later play these recordings back and share them among the team members.

### What this tutorial is not about

The basics of Java programming, and using [Chronon](#) are out of scope of this tutorial. Refer to the [Chronon documentation](#) for details.

### Before you start...

First, it is essential to understand that Chronon is not literally a debugger - it only helps you record the execution progress and then play it back, like a videotape.

Second, make sure that:

- You are working with IntelliJ IDEA version 13.1.
- The Chronon plugin is [downloaded and installed](#).

### Preparing an example

Let's see how Chronon works on a simple example of a two-thread class. One thread performs quick sorting, while the second thread performs bubble sorting.

First, create a project as described in the page [Creating and running your first Java application](#).

Next, [create a package](#) with the name demo, and, finally, [add Java classes](#) to this package. The first class is called ChrononDemo.java and it performs two-threaded array sorting:

```

package demo;

import org.jetbrains.annotations.NotNull;

import java.util.AbstractMap;
import java.util.Arrays;
import java.util.Random;

public class ChrononDemo {

    public static final int SIZE = 1000;
    public static final Random GENERATOR = new Random();

    public static void main(String[] args) throws InterruptedException {
        final int[] array = new int[SIZE];
        for (int i = 0; i < SIZE; i++) {
            array[i] = GENERATOR.nextInt();
        }
        final Thread quickSortThread = new Thread(new Runnable() {
            @Override
            public void run() {
                QuickSort.sort(Arrays.copyOf(array, array.length));
            }
        }, "Quick sort");

        final Thread bubbleThread = new Thread(new Runnable() {
            @Override
            public void run() {
                BubbleSort.sort(Arrays.copyOf(array, array.length));
            }
        }, "Bubble sort");

        quickSortThread.start();
        bubbleThread.start();

        quickSortThread.join();
        bubbleThread.join();
    }
}

```

The second is the class QuickSort.java that performs quick sorting:

```

package demo;

class QuickSort {
    private static int partition(int arr[], int left, int right) {
        int i = left, j = right;
        int tmp;
        int pivot = arr[(left + right) / 2];

        while (i <= j) {
            while (arr[i] < pivot)
                i++;
            while (arr[j] > pivot)
                j--;
            if (i <= j) {
                tmp = arr[i];
                arr[i] = arr[j];
                arr[j] = tmp;
                i++;
                j--;
            }
        }

        return i;
    }

    public static void sort(int arr[], int left, int right) {
        int index = partition(arr, left, right);
        if (left < index - 1)
            sort(arr, left, index - 1);
        if (index < right)
            sort(arr, index, right);
    }

    public static void sort(int arr[]) {
        sort(arr, 0, arr.length - 1);
    }
}

```

And, finally, the third one is the class BubbleSort.java that performs bubble sorting:

```


package demo;

public class BubbleSort {
    public static void sort(int[] arr) {
        boolean swapped = true;
        int j = 0;
        int tmp;
        while (swapped) {
            swapped = false;
            j++;
            for (int i = 0; i < arr.length - j; i++) {
                if (arr[i] > arr[i + 1]) {
                    tmp = arr[i];
                    arr[i] = arr[i + 1];
                    arr[i + 1] = tmp;
                    swapped = true;
                }
            }
        }
    }
}

```

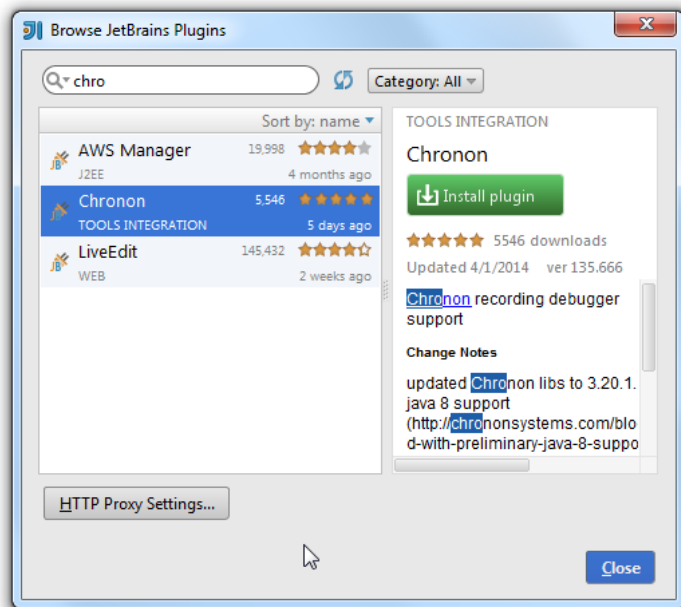
By the way, it is recommended to type the code manually, to see the magic IntelliJ IDEA's [code completion](#) in action.

## Installing plugin

Open the Settings/Preferences dialog. To do that, click  on the main toolbar, or press `Ctrl+Alt+S`. Under the IDE Settings, click the node `Plugins`.

The Chronon plugin is not bundled with IntelliJ IDEA, that's why you have to look for it in the JetBrains Plugins Repository. This is how it's done...

In the [Plugins page](#), click the button `Install JetBrains plugin...` to download and install plugins from the JetBrains repository. In the `Browse JetBrains Plugins` dialog box, find the Chronon plugin - you can type the search string in the filter area:



Install the plugin and restart IntelliJ IDEA for the changes to take effect.

## Changes to the UI

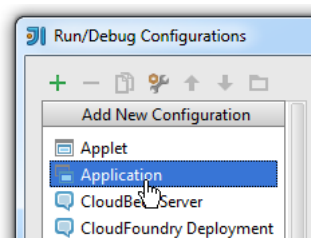
After restart, pay attention to the following changes:

- Dedicated `Run with Chronon` icon appears on the main toolbar. By now, this icon is disabled. It will become enabled as soon as the corresponding run/debug configuration appears.
- Chronon tool window (which becomes available on launching a run/debug configuration with Chronon, or on opening a Chronon record).
- Chronon tab appears in the run/debug configuration of the Application type (and some other types as well).
- `Run` menu is extended with two commands:
  - `Run <run/debug configuration name> with Chronon`
  - `Open Chronon recording`
- `Run <run/debug configuration name> with Chronon` and `Add logging statement` commands appear on the editor's context menu.

## Creating run/debug configuration

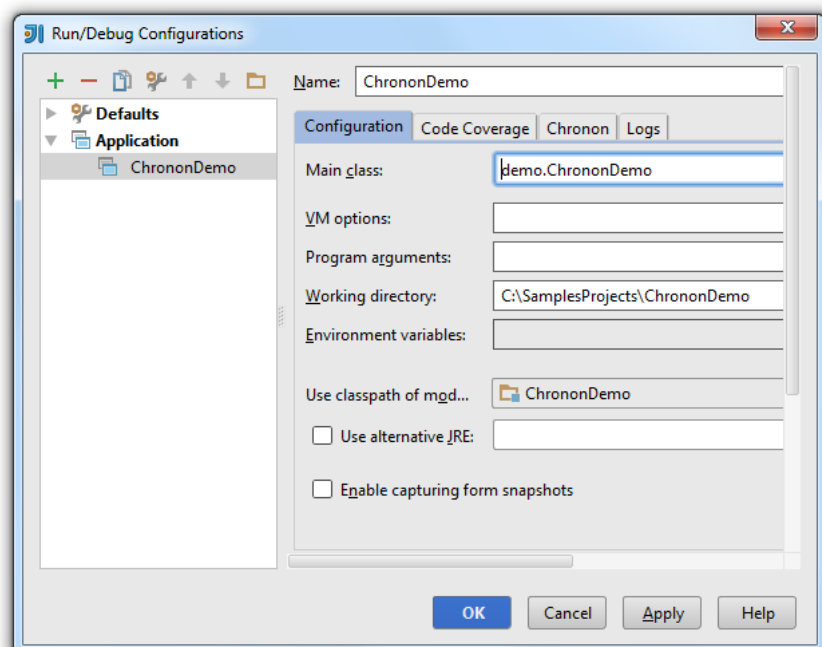
To launch our application, we need a run/debug configuration. Let's create one.

On the main menu, choose **Run**→**Edit Configuration** , and in the Run/Debug Configurations dialog box, click **+** . We are going to create a new run/debug configuration of the Application type, so select this type:



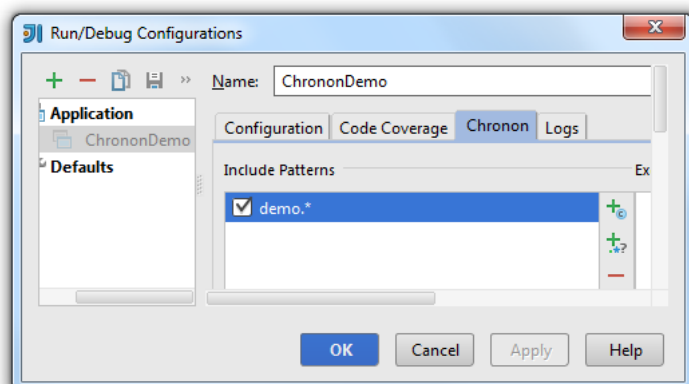
The new run/debug configuration based on the Application type appears. So far, it is unnamed and lacks reference to the class with the main method. Let's specify the missing information.

First, give this run/debug configuration a name. Let it be ChrononDemo. Next, press **Shift+Enter** and find the class with the main method ChrononDemo.java. This class resides in the package demo:



## Defining include/exclude patterns

Next, click the tab **Chronon** . In this tab, you have to specify which classes IntelliJ IDEA should look at. This is done by Include / Exclude Patterns:

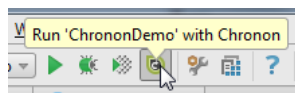


Now apply changes and close the dialog. The preliminary steps are ready.

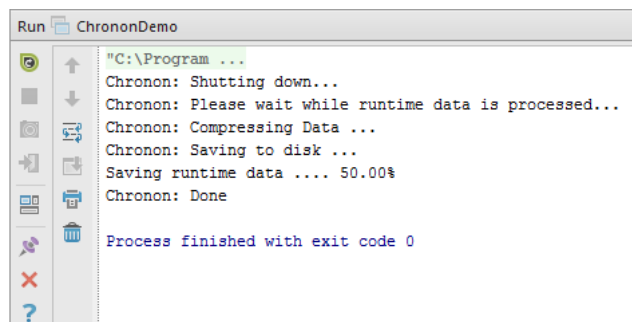
## Running with Chronon



OK, it's time to launch our application. To do that, either click the Chronon button on the main toolbar, or choose **Run→Run ChrononDemo with Chronon** on the main menu.

Let's choose the first way:

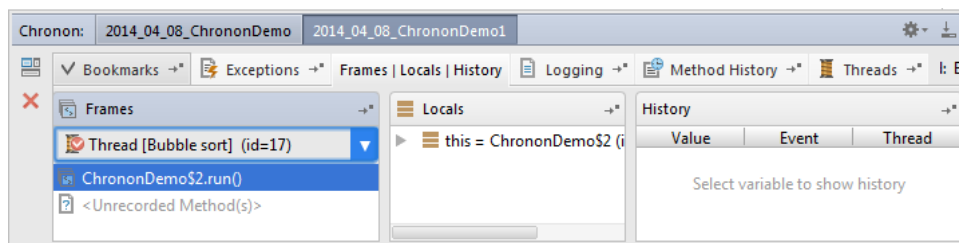


First thing that you see is the Run tool window that shows Chronon messages:



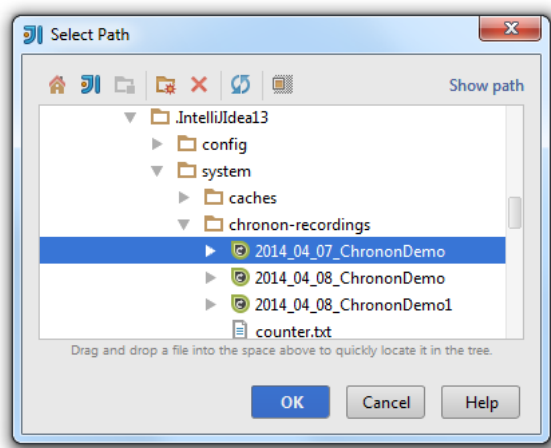
It is important to note that a Chronon record is NOT created, when you terminate your application by clicking . If it is necessary to stop an application and still have a Chronon record, click the Exit button  on the toolbar of the Run tool window.

Then the Chronon tool window appears - it looks very much like the Debug tool window. In this tool window you see a *record* created by Chronon; so doing, each record shows in its own tab:



Opening an existing record

By the way, if you want to open one of the previous records, use **Run→Open Chronon recording** on the main menu, and then choose the desired record:



## What can you do with a record?

In the Chronon tool window, you can:


Switch between threads

This is most easy - just switch to the Threads tab, and double-click the thread you are interested in. The selected thread is shown in boldface in the list of threads; besides that, the information about the currently selected thread appears in the upper-right part of the Chronon tool window:



Visual Studio Debugger - Threads Window

Name	Id	Priority	Start
main	1	5	40
Quick sort	16	5	24
Bubble sort	17	5	100

Note the progress bar. It shows the amount of passed events in the current thread:

Thread: main (1) Event: 183 

## Step through the application

Actually, you can use either the stepping commands of the **Run** menu, or the stepping buttons of the Chronon tool window. Unlike the debugger that allows only stepping forward, Chronon makes it possible to step through the applications in the reverse direction also. So, besides the traditional stepping buttons, there is **Step Backwards** button  and **Run Backwards to Cursor** button .

## Use bookmarks

Suppose you've stopped at a certain line, for example, in the main method of the class ChrononDemo.java:

```
ChrononDemo.java x
```

```
import java.util.AbstractMap;
import java.util.Arrays;
import java.util.Random;

public class ChrononDemo {

    public static final int SIZE = 1000;
    public static final Random GENERATOR = new Random();

    public static void main(String[] args) throws InterruptedException {
        final int[] array = new int[SIZE];
        for (int i = 0; i < SIZE; i++) {
            array[i] = GENERATOR.nextInt();
        }
    }
}
```


You want to memorize this place with its event number, to be able to return to it from any other location.

This is where the Bookmarks tab becomes helpful. Switch to this tab, and click . A bookmark for the current event, thread and method is created:

Event	Thread	Method	Description
7	main (1)	ChrononDemo.main(...)	

So doing, the bookmark memorizes the event number, thread and method name. Next, when you are in a different place of the code, clicking this bookmark in the Bookmarks tab will return you to this particular place.

## Explore methods

If you look at the editor, you notice the icons  in the left gutter next to the method declarations. Hovering the mouse pointer over such an icon shows a tooltip with the number of the recorded calls.

However, if you want to see a particular call event, then select the desired thread in the Threads tab (remember - the selected thread is shown in upper right part of the Chronon tool window), switch to the Method History tab, and track the execution history of a method:

Chronon 2014\_04\_10\_ChrononDemo

Bookmarks → Exceptions → Frames | Locals | History Logging → Method History → Threads → Thread: Quick sort (16)

History of QuickSort.sort(int[],int,int)

Call Event	Thread	arr	left	right	Return value
7	Quick sort (16)	(type ... 0	999		void
1238	Quick sort (16)	(type ... 0	351		void
1609	Quick sort (16)	(type ... 0	340		void
1972	Quick sort (16)	(type ... 0	326		void
2342	Quick sort (16)	(type ... 0	38		void

Thus you can explore the method execution, with the input and output data, which makes it easy to see how and where a method has been invoked.

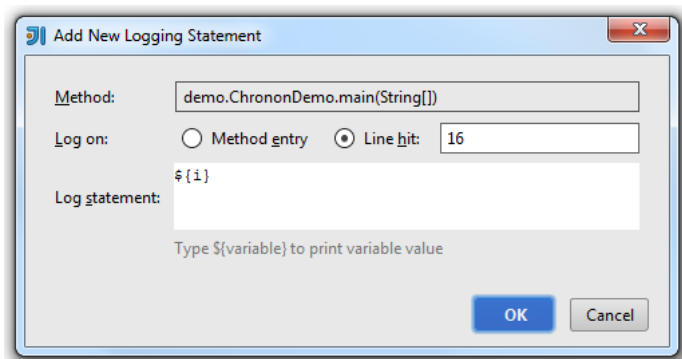
### Log values

What is the Logging tab for? By default, it is empty. However, using this tab makes it possible to define custom logging statements and track their output during application run.

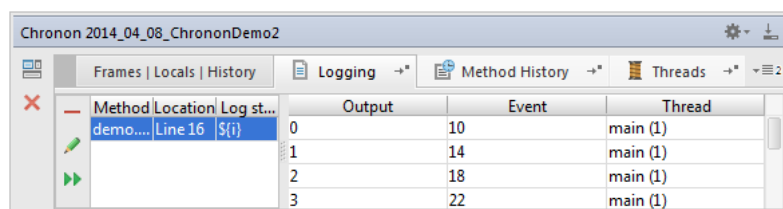
This is how logging works. In the editor, right-click the statement of interest, and choose

Add logging statement on the context menu. Then, in the dialog box that opens, specify the variable you want to watch in the format

```
${variable name}
```



Next, to make use of this logging statement, click  in the toolbar of the Logging tab. In the right-hand side of the Logging tab, the output of the logged statement is displayed:



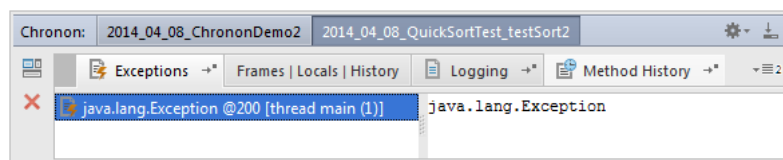
Note that such logging is equivalent to adding an output statement to your application; you could have added

```
System.out.println(<variable name>);
```

However, this would require application rebuild and rerun. With Chronon's logging facility, such complications are avoided.

### Explore exceptions

Suppose you want to find out how and when an exception has occurred. The Exceptions tab shows all exceptions that took place during the application execution. If you double click an exception, IntelliJ IDEA will bring you directly to the place of the exception occurrence:



## Summary

You've learned how to:

- Download and install Chronon plugin for IntelliJ IDEA
- Launch an application with Chronon recording.
- Open existing Chronon recordings.
- Work with the Chronon tool window. By the way, refer to the [Hide/Restore Toolbar](#) section of the Debug tool window reference to learn how to show/hide tabs.

This tutorial is over - congrats!

[Add Comment](#)