Standalone Application

Developer's Guide

MicroEJ 4.1

Reference:	TLT-0793-DGI-StandaloneApplicationDeveloperGuide-MicroEJ
Version:	4.1
Revision:	B

Revision History		
Revision 4.1-B	05/2017	
Removed board specific content. Updated some schematics.		
Revision 4.1-A	04/2017	
Updates for MicroEJ 4.1 and new schematics. Added a JUnit testsuite section.		
Revision A	07/2016	
Initial release		

Table of Contents

List of Figures

Chapter 1. MicroEJ Overview

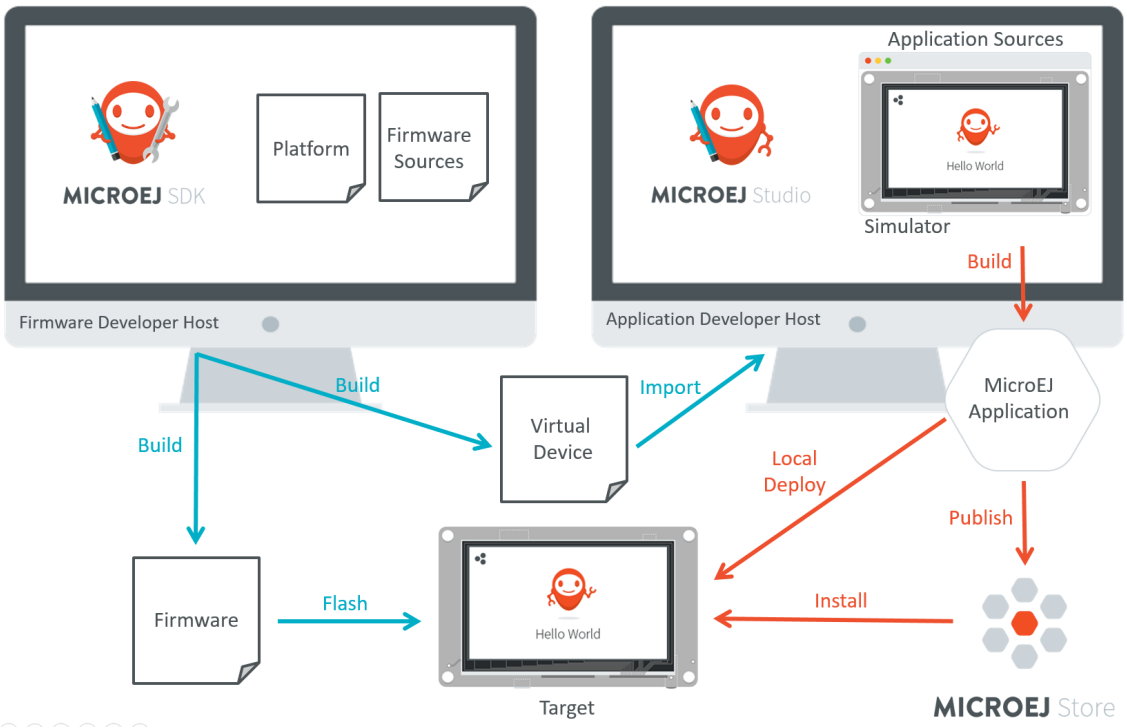
1.1. MicroEJ Editions

MicroEJ offers a comprehensive toolset to build the embedded software of a device. The toolset covers two levels in device software development:

- MicroEJ SDK for device firmware development
- MicroEJ Studio for application development

The firmware will generally be produced by the device OEM, it includes all device drivers and a specific set of MicroEJ functionalities useful for application developers targeting this device.

Figure 1.1. MicroEJ Development Tools Overview



Using the MicroEJ SDK tool, a firmware developer will produce two versions of the MicroEJ binary, each one able to run applications created with the MicroEJ Studio tool:

- A firmware binary to be flashed on OEM devices
- A Virtual Device which will be used as a device simulator by application developers

Using the MicroEJ Studio tool, an application developer will be able to:

- Import Virtual Devices matching his target hardware in order to develop and test applications on the simulator.
- Deploy the application locally on an hardware device equipped with the MicroEJ firmware

- Package and publish the application on a store, enabling remote end users to install it on their devices.

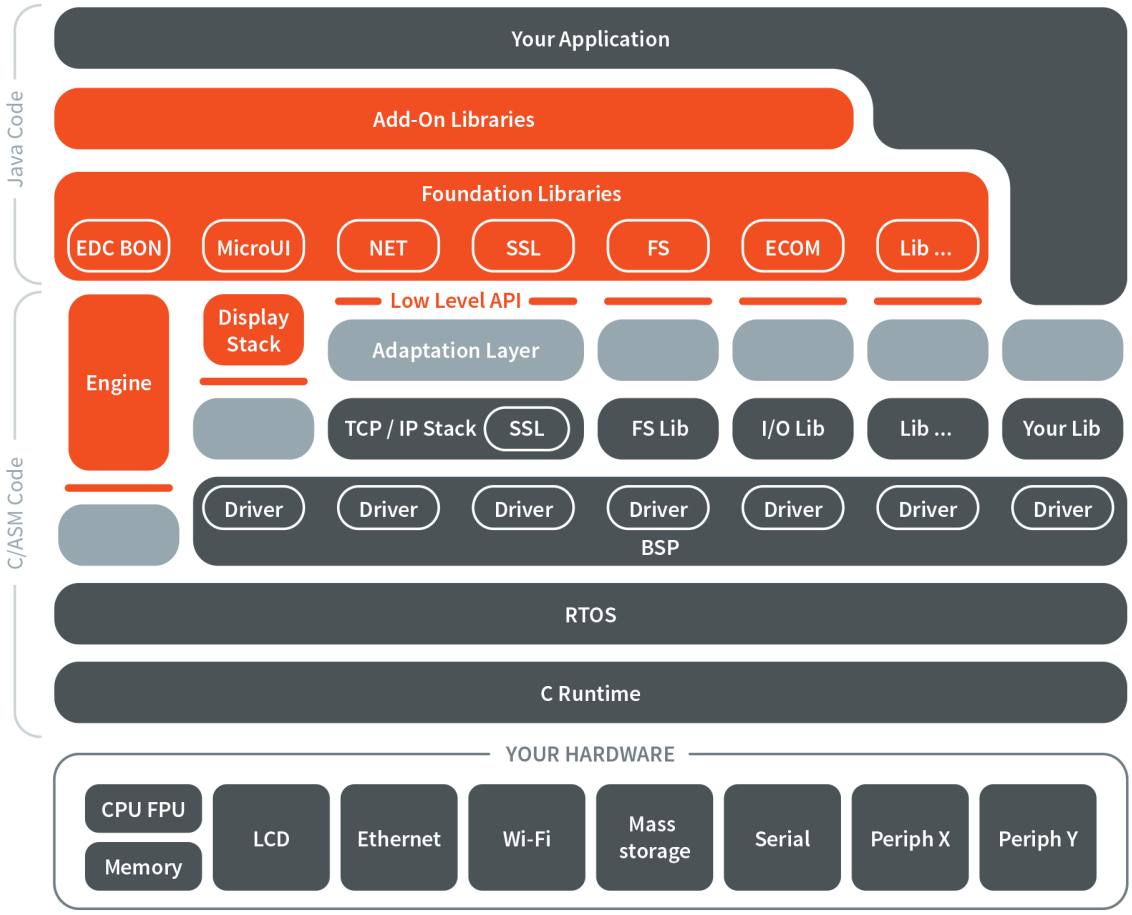
1.2. Firmware

1.2.1. Bootable Binary with Core Services

A MicroEJ firmware is a binary software program that can be programmed into the flash memory of a device. A MicroEJ firmware includes an instance of a MicroEJ runtime linked to:

- underlying native libraries and BSP + RTOS,
- MicroEJ libraries and application code (C and Java code).

Figure 1.2. MicroEJ Firmware Architecture



1.2.2. Specification

The set of libraries included in the firmware and its dimensioning limitations (maximum number of simultaneous threads, open connections, ...) are firmware specific. Please refer to `http://developer.microej.com/getting-started.html` firmware release notes.

Chapter 2. MicroEJ SDK Getting Started

2.1. Introducing MicroEJ SDK

MicroEJ SDK provides tools based on Eclipse to develop software applications for MicroEJ-ready devices. MicroEJ SDK allows application developers to write MicroEJ applications and run them on a virtual (simulated) or real device.

This document is a step-by-step introduction to application development with MicroEJ SDK. The purpose of MicroEJ SDK is to develop for targeted MCU/MPU computers (IoT, wearable, etc.) and it is therefore a cross-development tool.

Unlike standard low-level cross-development tools, MicroEJ SDK offers unique services like hardware simulation and local deployment to the target hardware.

Application development is based on the following elements:

- MicroEJ SDK, the integrated development environment for writing applications. It is based on Eclipse and is relies on the integrated Java compiler (JDT). It also provides a dependency manager for managing MicroEJ Libraries (see Section 3.5, "Library Dependency Manager"). The current version of MicroEJ SDK is built on top of Eclipse Mars (`http://www.eclipse.org/downloads/packages/release/Mars/2`).
- MicroEJ Platform, a software package including the resources and tools required for building and testing an application for a specific MicroEJ-ready device. MicroEJ platforms are imported into MicroEJ SDK within a local folder called MicroEJ Platforms repository. Once a MicroEJ platform is imported, an application can be launched and tested on simulator. It also provides a means to locally deploy the application on a MicroEJ-ready device.
- MicroEJ-ready device, an hardware device that will be programmed with a MicroEJ firmware. A MicroEJ firmware is a binary instance of MicroEJ runtime for a target hardware board.

Starting from scratch, the steps to go through the whole process are detailed in the following sections of this chapter :

- Download and install a MicroEJ Platform
- Build and run your first application on simulator
- Build and run your first application on target hardware

2.2. Setup MicroEJ SDK

2.2.1. Download and Install a MicroEJ Platform

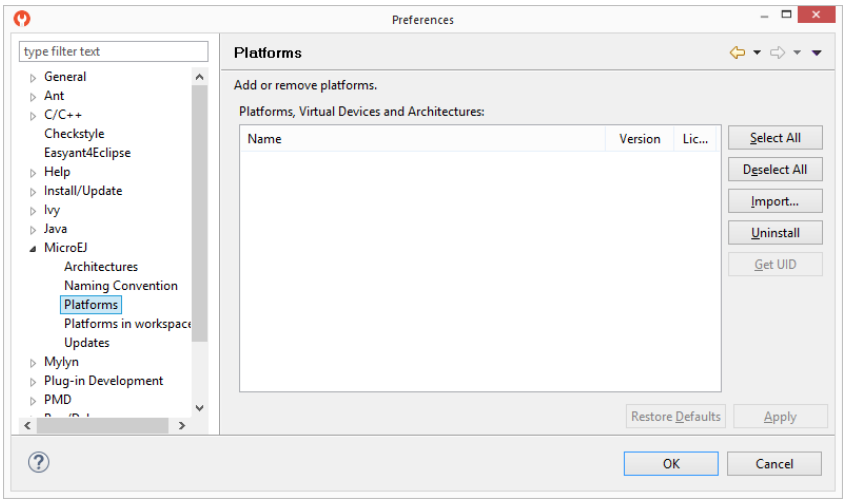
MicroEJ SDK being a cross development tool, it does not build software targeted to your host desktop platform. In order to run MicroEJ applications, a target hardware is required. Several commercial targets boards from main MCU/MPU chip manufacturers can be prepared to run MicroEJ applications, you can also run your applications without one of these boards with the help of a MicroEJ Simulator.

A MicroEJ Platform is a software package including the resources and tools required for building and testing an application for a specific MicroEJ-ready device. MicroEJ Platforms are available at `http://developer.microej.com/getting-started.html`.

After downloading the MicroEJ Platfom, launch MicroEJ SDK on your desktop to start the process of Platform installation :

- Open the Platform view in MicroEJ SDK, select `Window > Preferences > MicroEJ > Platforms`. The view should be empty on a fresh install of the tool

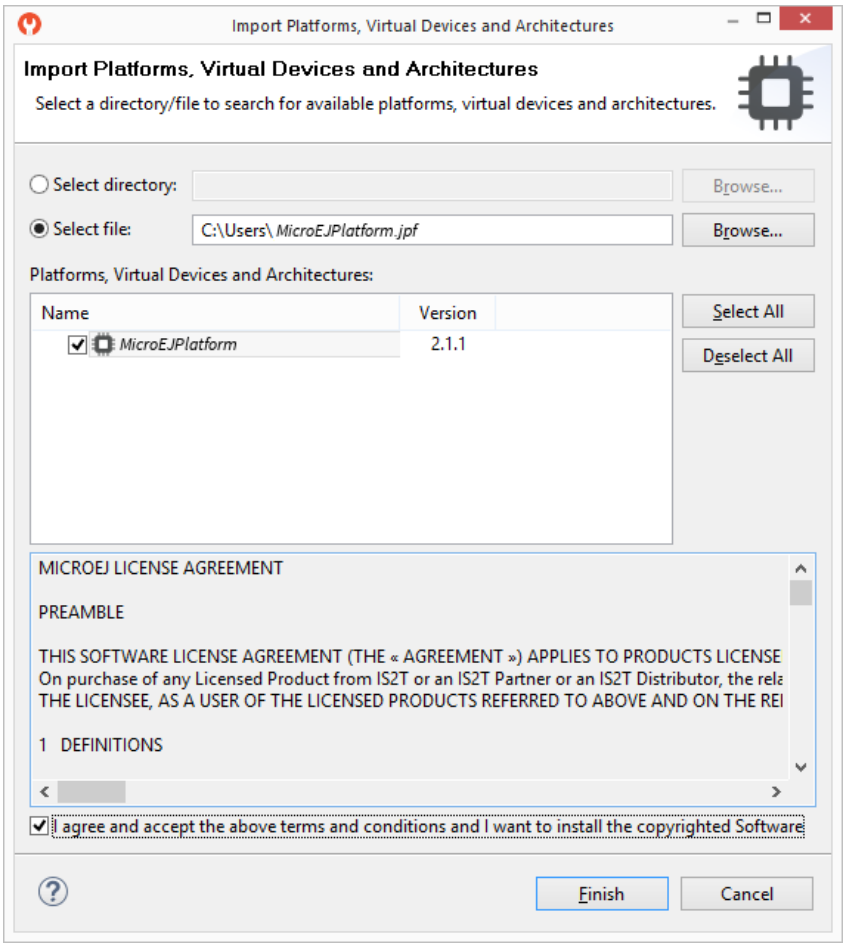
Figure 2.1. MicroEJ Platform Import



- Press `Import...` button.

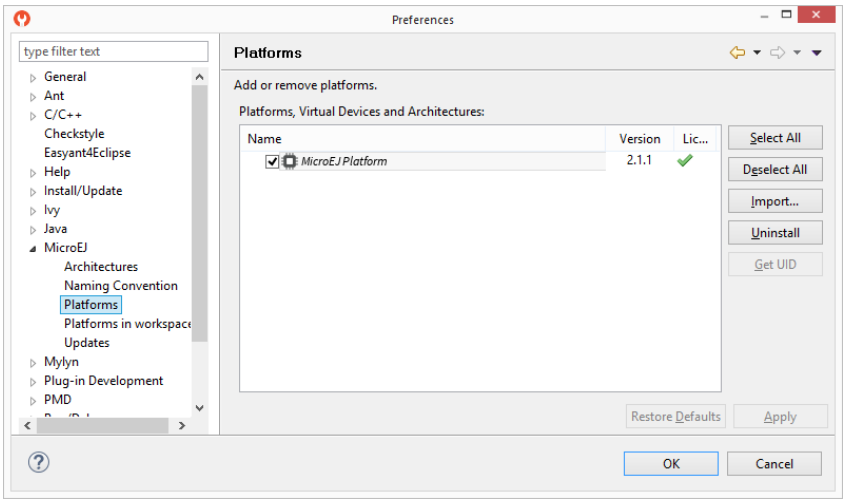
- Choose `Select File...` and use the `Browse` option to navigate to the `.jpf` file containing your MicroEJ Platform, then read and accept the license agreement to proceed.

Figure 2.2. MicroEJ Platform Selection



- The MicroEJ Platform should now appear in the `Platforms` view, with a green valid mark.

Figure 2.3. MicroEJ Platform List



2.2.2. Setup Ivy Repository

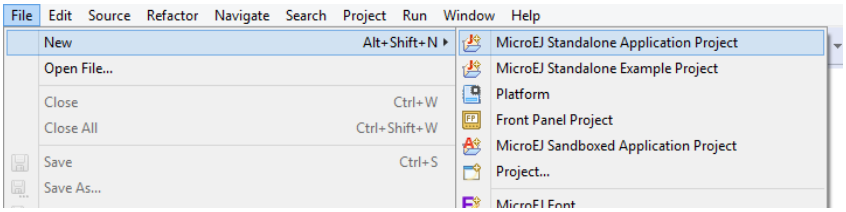
MicroEJ SDK uses an Ivy repository which provides a set of libraries. MicroEJ SDK is already configured to use an online Ivy repository. For offline use, you can download the offline repository available on website `http://developer.microej.com/4.0/ivy/`, and follow the described steps to use this repository.

2.3. Build and Run an Application

2.3.1. Create a MicroEJ Standalone Application

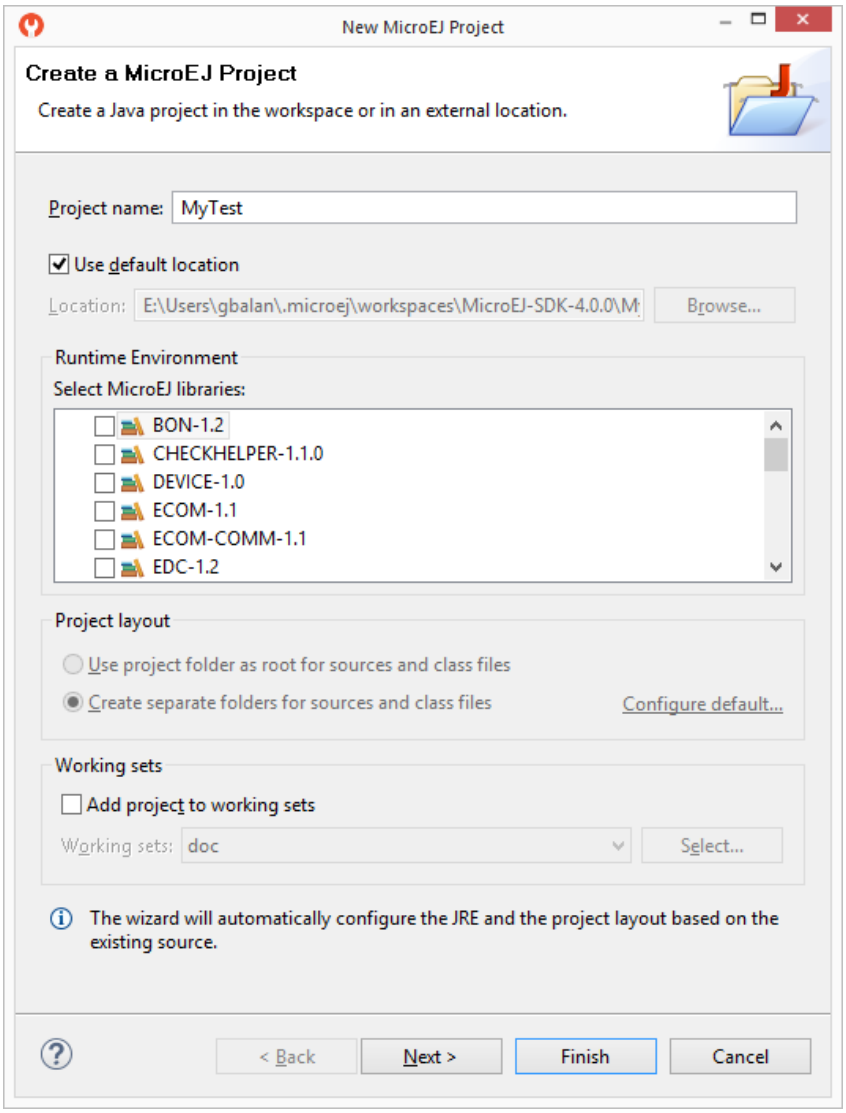
- Create a project in your workspace. Select `File > New > MicroEJ Standalone Application Project`.

  Figure 2.4. New MicroEJ Standalone Application Project



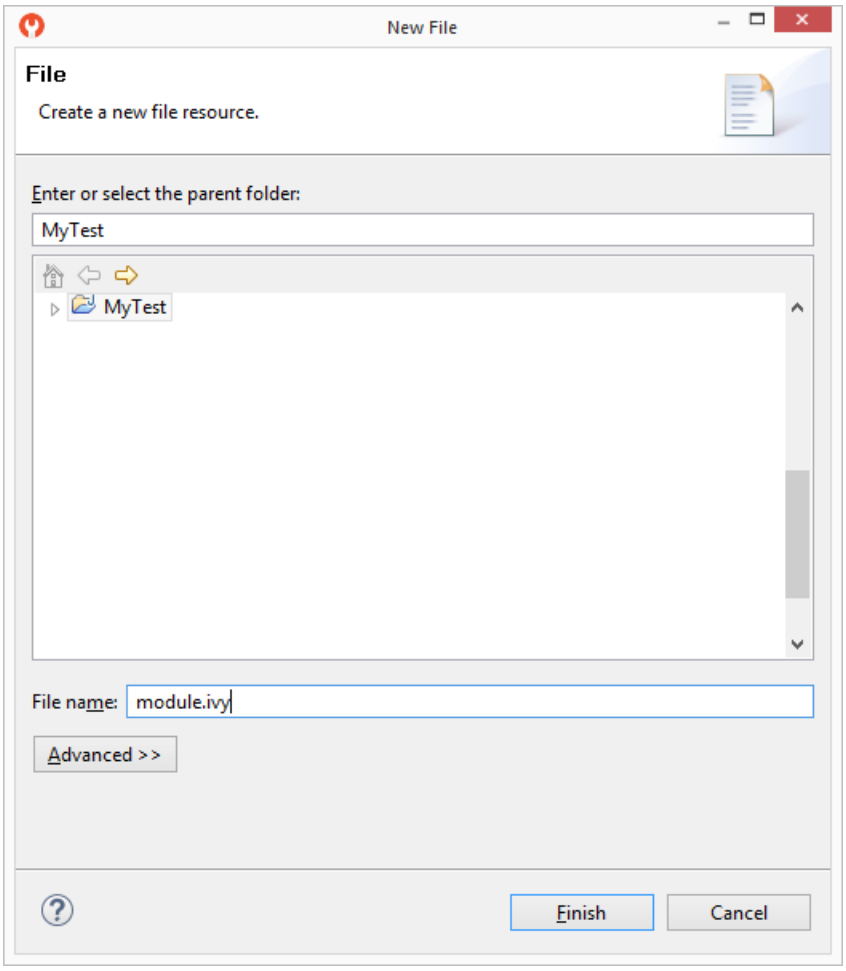
- In the `New MicroEJ Project` window, fill `Project name` and uncheck the MicroEJ library `EDC-1.2` in the `Runtime Environment` list. Click on `Finish`. You now have an application project created in MicroEJ SDK.

Figure 2.5. MicroEJ Standalone Application Project Configuration



- Next step consists in specifying the classpath of your application using Ivy dependencies. Right click on the project and select `New > File`. Fill the `File name: module.ivy`. Click on `Finish`.

Figure 2.6. New Ivy File

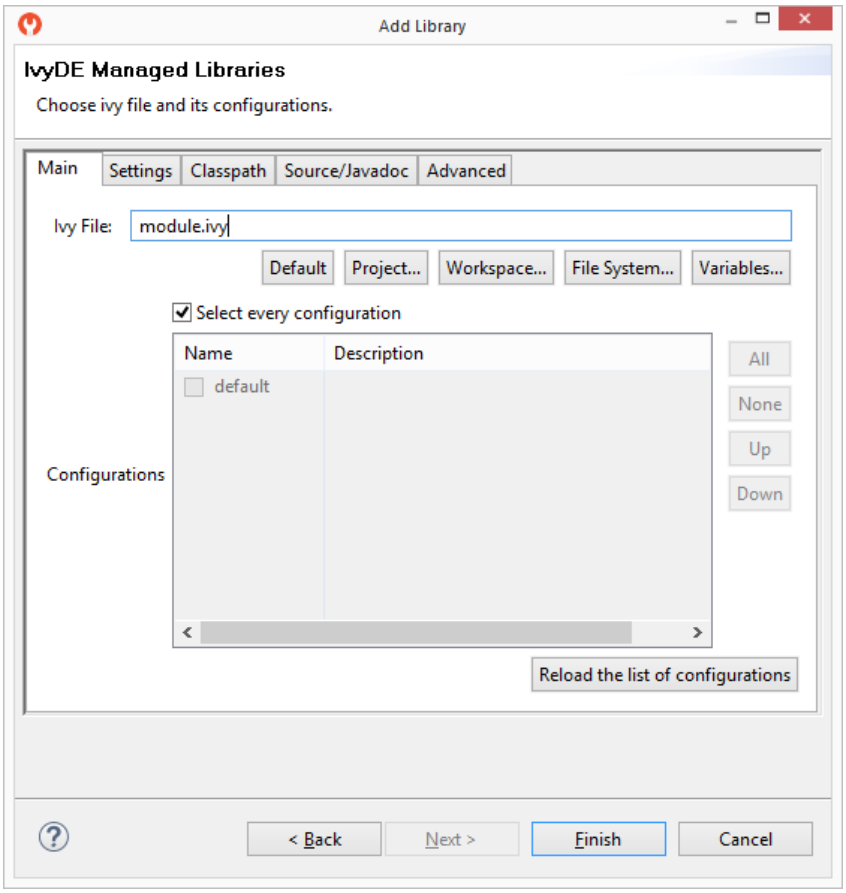


- The Eclipse internal text editor allows you to edit the new created file. Copy / paste the following lines and save the file:

```
<ivy-module version="2.0">
 <info module=""/>
 <dependencies>
  <dependency org="ej.api" name="edc" rev="1.2.+"/>
  <dependency org="ej.api" name="bon" rev="1.2.+"/>
 </dependencies>
</ivy-module>
```

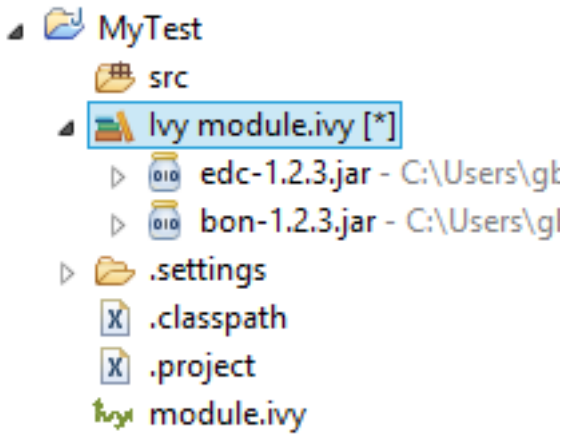
- Right click on the project and select `Build Path > Add Libraries`. Select `IvyDE Managed Dependencies` and click on `Next`. Fill the `Ivy File` name: `module.ivy`. Finally, click on `Finish` and on `OK`.

Figure 2.7. MicroEJ Application Build Path



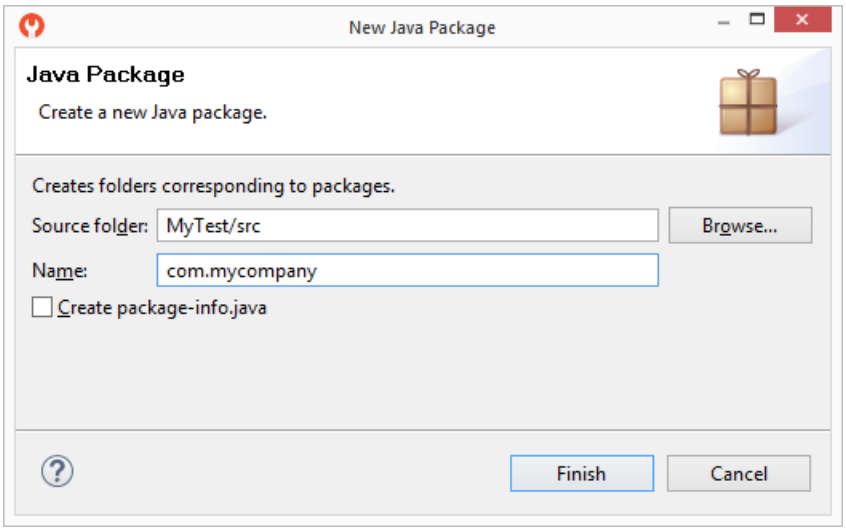
The project now uses some Ivy dependencies. You can see them opening the `Ivy` tree:

Figure 2.8. MicroEJ Application Dependencies



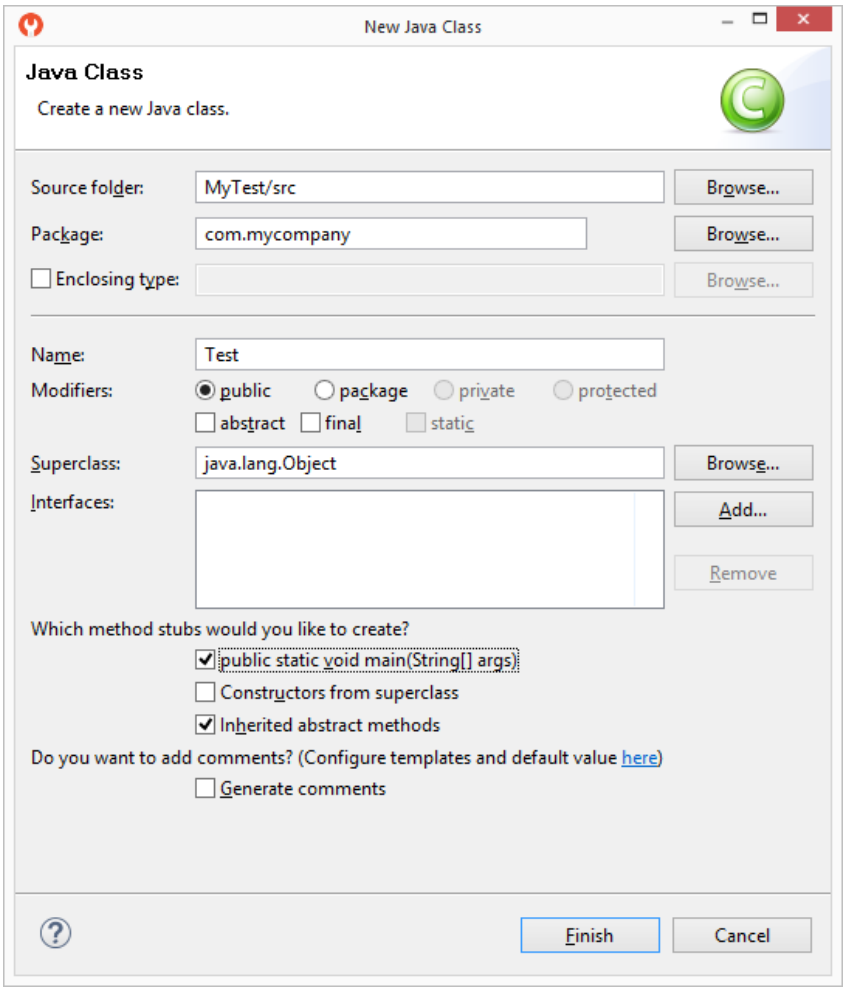
- Right click on the source folder `src` and select `New > Package`. Give a name: `com.mycompany`. Click on `Finish`.

Figure 2.9. New Package



- The package `com.mycompany` is available under `src` folder. Right click on this package and select `New > Class`. Give a name: `Test` and check the box `public static void main(String[] args)`. Click on `Finish`.

Figure 2.10. New Class

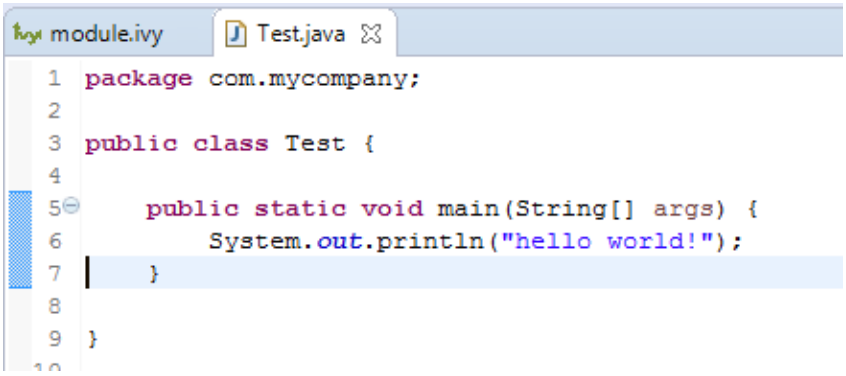


- The new class has been created with an empty `main()` method. Fill the method body with the following lines:

```
System.out.println("hello world!");
```

.

Figure 2.11. MicroEJ Application Content

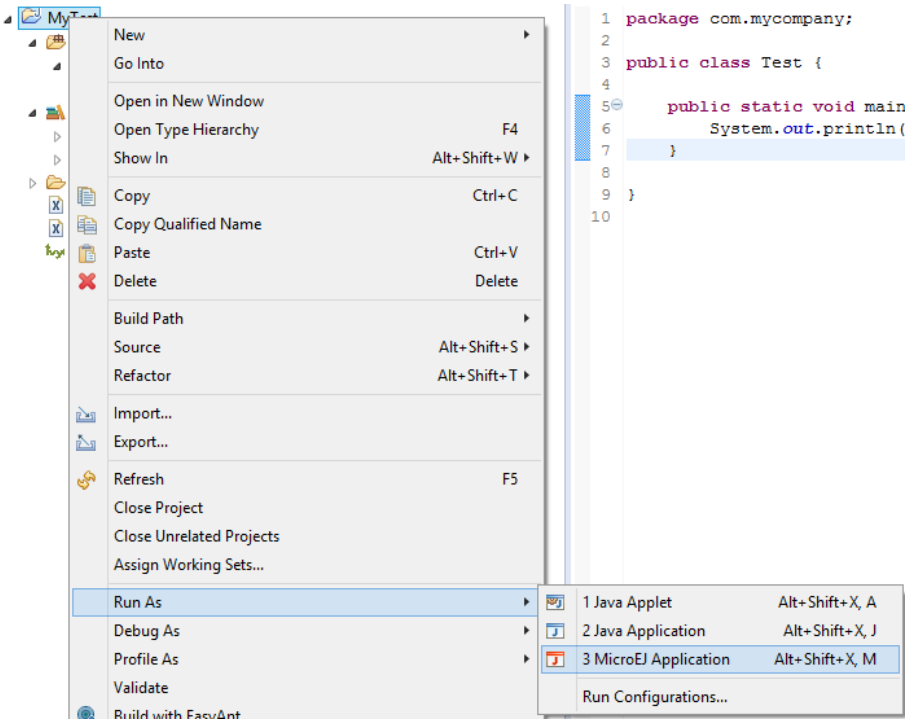


The test application is now ready to be executed. See next chapters.

2.3.2. Run on the Simulator

To run the sample project on Simulator, select it in the left panel then right-click and select Run > `Run as > MicroEJ Application`.

Figure 2.12. MicroEJ Development Tools Overview



MicroEJ SDK console will display Launch steps messages.

```
=============== [ Initialization Stage ] ===============
=============== [ Launching on Simulator ] ===============
hello world!
=============== [ Completed Successfully ] ===============

SUCCESS
```

2.3.3. Run on the Hardware Device

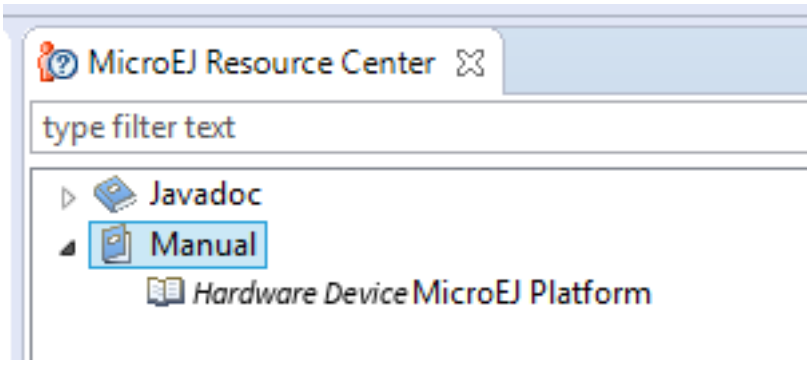
Compile an application, connect the hardware device and deploy on it is hardware dependant. These steps are described in dedicated documentation available inside the MicroEJ Platform. This documentation is accessible from the MicroEJ Resources Center view.

Note

MicroEJ Resources Center view may have been closed. Click on `Help > MicroEJ Resources Center` to reopen it.

Open the menu `Manual` and select the documentation `[hardware device] MicroEJ Platform`, where `[hardware device]` is the name of the hardware device. This documentation features a guide to run a built-in application on MicroEJ Simulator and on hardware device.

Figure 2.13. MicroEJ Platform Guide



2.4. Application Development

The following sections of this document shall prove useful as a reference when developing applications for MicroEJ. They cover concepts essential to MicroEJ applications design.

In addition to these sections, by going to `http://developer.microej.com/`, you can access a number of helpful resources such as:

- Libraries,

- Application Examples, with their source code,
- Documentation (HOWTOs, Reference Manuals, APIs javadoc...)

Chapter 3. MicroEJ Classpath

MicroEJ applications run on a target device and their footprint is optimized to fulfill embedded constraints. The final execution context is an embedded device that may not even have a file system. Files required by the application at runtime are not directly copied to the target device, they are compiled to produce the application binary code which will be executed by MicroEJ core engine.

As a part of the compile-time trimming process, all types not required by the embedded application are eliminated from the final binary.

MicroEJ Classpath is a developer defined list of all places containing files to be embedded in the final application binary. MicroEJ Classpath is made up of an ordered list of paths. A path is either a folder or a zip file, called a JAR file (JAR stands for Java ARchive).

- Section 3.1, “Application Classpath” explains how the MicroEJ classpath is built from a MicroEJ application project.
- Section 3.2, “Classpath Load Model” explains how the application content is loaded from MicroEJ Classpath.
- Section 3.3, “Classpath Elements” specifies the different elements that can be declared in MicroEJ Classpath to describe the application content.
- Section 3.4, “Foundation vs Add-On Libraries” explains the different kind of libraries that can be added to MicroEJ Classpath.
- Finally, Section 3.5, “Library Dependency Manager” shows how to manage libraries dependencies in MicroEJ.

3.1. Application Classpath

The following schema shows the classpath mapping from a MicroEJ application project to the MicroEJ Classpath ordered list of folders and JAR files. The classpath resolution order (left to right) follows the project appearance order (top to bottom).

Figure 3.1. MicroEJ Application Classpath Mapping

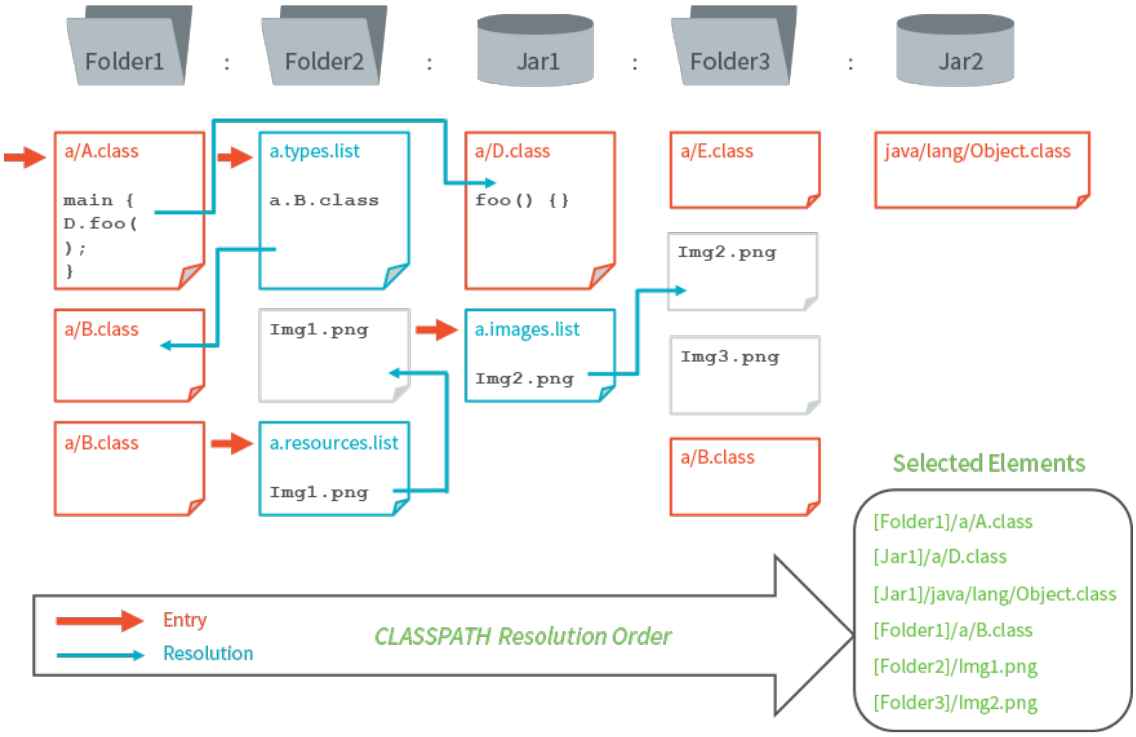
3.2. Classpath Load Model

A MicroEJ Application classpath is created via the loading of :

- an entry point type
- all `*.[extension].list` files declared in a MicroEJ Classpath.

The different elements that constitute an application are described in Section 3.3, “Classpath Elements”. They are searched within MicroEJ Classpath from left to right (the first file found is loaded). Types referenced by previously loaded MicroEJ Classpath elements are loaded transitively.

Figure 3.2. Classpath Load Principle



3.3. Classpath Elements

The MicroEJ Classpath contains the following elements:

- An entrypoint described in section Section 3.3.1, “Application Entry Points”
- Types in `.class` files, described in section Section 3.3.2, “Types”
- Raw resources, described in section Section 3.3.3, “Raw Resources”
- Immutables Object data files, described in Section Section 3.3.4, “Immutable Objects”
- Images and Fonts resources
- `*.[extension].list` files, declaring contents to load. Supported list file extensions and format is specific to declared application content and is described in the appropriate section.

3.3.1. Application Entry Points

MicroEJ application entry point is a class that contains a `public static void main(String[])` method. In case of MicroEJ Sandboxed Application, this entry point is automatically generated by MicroEJ Studio from declared Activity and/or BackgroundService types. In case of a MicroEJ Standalone application, this has to be defined by the user.

3.3.2. Types

MicroEJ types (classes, interfaces) are compiled from source code (`.java`) to classfiles (`.class`). When a type is loaded, all types dependencies found in the classfile are loaded (transitively).

A type can be declared as a *Required type* in order to enable the following usages:

- to be dynamically loaded from its name (with a call to `Class.forName(String)`)
- to retrieve its fully qualified name (with a call to `Class.getName()`)

A type that is not declared as a *Required type* may not have its fully qualified name (FQN) embedded. Its FQN can be retrieved using the stack trace reader tool (see Section 4.3, “Stack Trace Reader”).

Required Types are declared in MicroEJ Classpath using `*.types.list` files. The file format is a standard Java properties file, each line listing the fully qualified name of a type. Example:

Example 3.1. Required Types **`*.types.list`** File Example

```
# The following types are marked as MicroEJ Required Types
com.mycompany.MyImplementation
java.util.Vector
```

3.3.3. Raw Resources

Raw resources are binary files that need to be embedded by the application so that they may be dynamically retrieved with a call to `Class.getResourceAsStream(java.io.InputStream)`. Raw Resources are declared in MicroEJ Classpath using `*.resources.list` files. The file format is a standard Java properties file, each line is a relative `/` separated name of a file in MicroEJ Classpath to be embedded as a resource. Example:

Example 3.2. Raw Resources **`*.resources.list`** File Example

```
# The following resource is embedded as a raw resource
com/mycompany/MyResource.txt
```

3.3.4. Immutable Objects

Immutables objects are regular read-only objects that can be retrieved with a call to `ej.bon.Immutables.get(String)`. Immutables objects are declared in files called *immutable objects data files*, which format is described in the `[B-ON]` specification (`http://e-s-r.net`). Immutables objects data files are declared in MicroEJ Classpath using `*.immutables.list` files. The file format is a standard Java properties file, each line is a `/` separated name of a relative file in MicroEJ Classpath to be loaded as an Immutable objects data file. Example:

Example 3.3. Immutable Objects Data Files **`*.immutables.list`** File Example

```
# The following file is loaded as an Immutable objects data files
com/mycompany/MyImmutables.data
```

3.3.5. System Properties

System Properties are key/value string pairs that can be accessed with a call to `System.getProperty(String)`. System properties are declared in MicroEJ Classpath `*.properties.list` files. The file format is a standard Java properties file. Example:

Example 3.4. System Properties **`*.properties.list`** File Example

```
# The following property is embedded as a System property
com.mycompany.key=com.mycompany.value
```

3.3.6. Images

3.3.6.1. Overview

Images are graphical resources that can be accessed with a call to `ej.microui.display.Image.createImage()`. To be displayed, these images have to be converted from their source format to the display raw format. The conversion can either be done at :

- build-time (using the image generator tool)
- run-time (using the relevant decoder library)

Images that must be processed by the image generator tool are declared in MicroEJ Classpath `*.images.list` files. The file format is a standard Java properties file, each line representing a `/` separated resource path relative to the MicroEJ classpath root referring to a standard image file (e.g. `.png`, `.jpg`). The resource may be followed by an optional parameter (separated by a `:`) which defines and/or describe the image output file format (raw format). When no option is specified, the image is embedded as-is and will be decoded at run-time (although listing files without format specifier has no impact on the image generator processing, it is advised to specify them in the `*.images.list` files anyway, as it makes the run-time processing behavior explicit). Example:

Figure 3.3. Image Generator **`*.images.list`** File Example

```
# The following image is embedded
# as a PNG resource (decoded at run-time)
com/mycompany/MyImage1.png

# The following image is embedded
# as a 16 bits format without transparency (decoded at build-time)
com/mycompany/MyImage2.png:RGB565

# The following image is embedded
# as a 16 bits format with transparency (decoded at build-time)
com/mycompany/MyImage3.png:ARGB1555
```

3.3.6.2. Output Formats

3.3.6.2.1. No Compression

When no output format is set in the images list file, the image is embedded without any conversion / compression. This allows you to embed the resource as well, in order to keep the source image characteristics (compression, bpp etc.). This option produces the same result as specifiying an image as a resource in the MicroEJ launcher.

Advantages:

- Preserves the image characteristics.

Disadvantages:

- Requires an image runtime decoder.
- Requires some RAM in which to store the decoded image

Figure 3.4. Unchanged Image Example

```
image1
```

3.3.6.2.2. Display Output Format

This format encodes the image into the exact display memory representation. If the image to encode contains some transparent pixels, the output file will embed the transparency according to the display's implementation capacity. When all pixels are fully opaque, no extra information will be stored in the output file in order to free up some memory space.

Advantages:

- Drawing an image is very fast.
- Supports alpha encoding.

Disadvantages:

- No compression: the image size in bytes is proportional to the number of pixels.

Figure 3.5. Display Output Format Example

```
image1:display
```

3.3.6.2.3. Generic Output Formats

Depending on the target hardware, several generic output formats are available. Some formats may be directly managed by the BSP display driver. Refer to the platform specification to retrieve the list of natively supported formats.

Advantages:

- The pixels layout and bits format are standard, so it is easy to manipulate these images on the C-side.
- Drawing an image is very fast when the display driver recognizes the format (with or without transparency).
- Supports or not the alpha encoding: select the most suitable format for the image to encode.

Disadvantages:

- No compression: the image size in bytes is proportional to the number of pixels, the transparency, and the bits-per-pixel.

Select one the following format to use a generic format:

- ARGB8888: 32 bits format, 8 bits for transparency, 8 per color.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return c;
}
```

- RGB888: 24 bits format, 8 per color. Image is always fully opaque.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return c & 0xffffff;
}
```

- ARGB4444: 16 bits format, 4 bits for transparency, 4 per color.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return 0
   | ((c & 0xf0000000) >> 16)
   | ((c & 0x00f00000) >> 12)
   | ((c & 0x0000f000) >> 8)
   | ((c & 0x000000f0) >> 4)
   ;
}
```

- ARGB1555: 16 bits format, 1 bit for transparency, 5 per color.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return 0
   | (((c & 0xff000000) == 0xff000000) ? 0x8000 : 0)
   | ((c & 0xf80000) >> 9)
   | ((c & 0x00f800) >> 6)
   | ((c & 0x0000f8) >> 3)
   ;
}
```

- RGB565: 16 bits format, 5 or 6 per color. Image is always fully opaque.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return 0
   | ((c & 0xf80000) >> 8)
   | ((c & 0x00fc00) >> 5)
   | ((c & 0x0000f8) >> 3)
   ;
}
```

- A8: 8 bits format, only transparency is encoded. The color to apply when drawing the image, is the current GraphicsContext color.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return 0xff - (toGrayscale(c) & 0xff);
}
```

- A4: 4 bits format, only transparency is encoded. The color to apply when drawing the image, is the current GraphicsContext color.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return (0xff - (toGrayscale(c) & 0xff)) / 0x11;
}
```

- A2: 2 bits format, only transparency is encoded. The color to apply when drawing the image, is the current GraphicsContext color.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return (0xff - (toGrayscale(c) & 0xff)) / 0x55;
}
```

- A1: 1 bit format, only transparency is encoded. The color to apply when drawing the image, is the current GraphicsContext color.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return (0xff - (toGrayscale(c) & 0xff)) / 0xff;
}
```

- C4: 4 bits format with grayscale conversion. Image is always fully opaque.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return (toGrayscale(c) & 0xff) / 0x11;
}
```

- C2: 2 bits format with grayscale conversion. Image is always fully opaque.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return (toGrayscale(c) & 0xff) / 0x55;
}
```

- C1: 1 bit format with grayscale conversion. Image is always fully opaque.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return (toGrayscale(c) & 0xff) / 0xff;
}
```

- AC44: 4 bits for transparency, 4 bits with grayscale conversion.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return 0
  | ((color >> 24) & 0xf0)
  | ((toGrayscale(color) & 0xff) / 0x11)
  ;
}
```

- AC22: 2 bits for transparency, 2 bits with grayscale conversion.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return 0
  | ((color >> 28) & 0xc0)
  | ((toGrayscale(color) & 0xff) / 0x55)
  ;
}
```

- AC11: 1 bit for transparency, 1 bit with grayscale conversion.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return 0
  | ((c & 0xff000000) == 0xff000000 ? 0x2 : 0x0)
  | ((toGrayscale(color) & 0xff) / 0xff)
  ;
}
```

Figure 3.6. Generic Output Format Examples

```
image1:ARGB8888
image2:RGB565
image3:A4
```

3.3.6.2.4. RLE1 Output Format

The image engine can display embedded images that are encoded into a compressed format which encodes several consecutive pixels into one or more 16-bits words. This encoding manages a maximum alpha level of 2 (alpha level is always assumed to be 2, even if the image is not transparent).

- Several consecutive pixels have the same color (2 words).
    - First 16-bit word specifies how many consecutive pixels have the same color.
    - Second 16-bit word is the pixels' color.
- Several consecutive pixels have their own color (1 + n words).
    - First 16-bit word specifies how many consecutive pixels have their own color.
    - Next 16-bit word is the next pixel color.
- Several consecutive pixels are transparent (1 word).
    - 16-bit word specifies how many consecutive pixels are transparent.

Advantages:

- Supports 0 & 2 alpha encoding.
- Good compression when several consecutive pixels respect one of the three previous rules.

Disadvantages:

- Drawing an image is slightly slower than when using Display format.

Figure 3.7. RLE1 Output Format Example

```
image1:RLE1
```

3.3.7. Fonts

3.3.7.1. Overview

Fonts are graphical resources that can be accessed with a call to `ej.microui.display.Font.getFont()`. To be displayed, these fonts have to be converted at build-time from their source format to the display raw format by the font generator tool. Fonts that must be processed by the font generator tool are declared in MicroEJ Classpath `*.fonts.list` files.

The file format is a standard Java properties file, each line representing a `/` separated resource path relative to the MicroEJ classpath root referring to a MicroEJ font file (usually with a `.ejf` file extension). The resource may be followed by optional parameters which define :

- some ranges of characters to embed in the final raw file
- the required pixel depth for transparency.

By default, all characters available in the input font file are embedded, and the pixel depth is `1` (i.e 1 bit-per-pixel). Example:

Figure 3.8. Font Generator **`*.fonts.list`** File Example

```
# The following font is embedded with all characters
# without transparency
com/mycompany/MyFont1.ejf

# The following font is embedded with only the latin
# unicode range without transparency
com/mycompany/MyFont2.ejf:latin

# The following font is embedded with all characters
# with 2 levels of transparency
com/mycompany/MyFont2.ejf::2
```

MicroEJ font files conventionally end with the `.ejf` suffix and are created using the Font Designer (see Section 4.2, “Font Designer”).

3.3.7.2. Font Range

The first parameter is for specifying the font ranges to embed. Selecting only a specific set of characters to embed reduces the memory footprint. Several ranges can be specified, separated by `;`. There are two ways to specify a character range: the custom range and the known range.

3.3.7.2.1. Custom Range

Allows the selection of raw Unicode character ranges.

Examples:

- `myfont:0x21-0x49`: Embed all characters from 0x21 to 0x49 (included).
- `myfont:0x21-0x49,0x55`: Embed all characters from 0x21 to 0x49 and character 0x55
- `myfont:0x21-0x49;0x55`: Same as previous, but done by declaring two ranges.

3.3.7.2.2. Known Range

A known range is a range defined by the "Unicode Character Database" version 9.0.0 available on `http://www.unicode.org/`. Each range is composed of sub ranges that have a unique id.

- `myfont:basic_latin`: Embed all *Basic Latin* characters.

- `myfont:basic_latin;arabic`: Embed all *Basic Latin* characters, and all *Arabic* characters.

3.3.7.3. Transparency

The second parameter is for specifying the font transparency level (`1`, `2`, `4` or `8`).

Examples:

- `myfont:latin:4`: Embed all latin characters with 4 levels of transparency
- `myfont::2`: Embed all characters with 2 levels of transparency

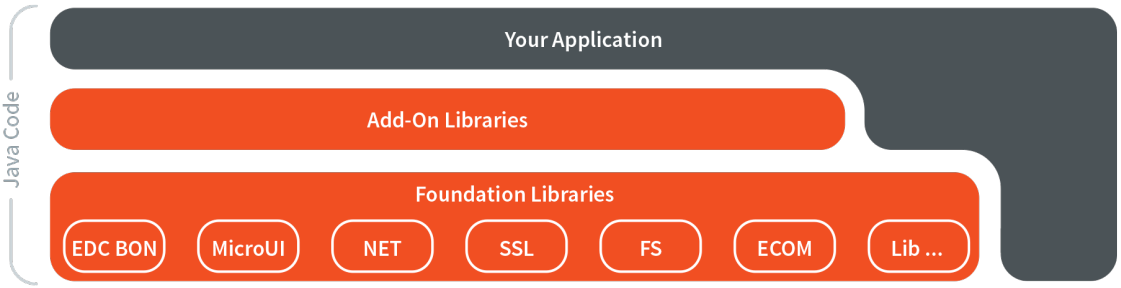
3.4. Foundation vs Add-On Libraries

A MicroEJ Foundation Library is a MicroEJ Core library that provides core runtime APIs or hardware-dependent functionality. A Foundation library is divided into an API and an implementation. A Foundation library API is composed of a name and a 2 digits version (e.g. `EDC-1.2`, `MWT-2.0`) and follows the semantic versioning (`http://semver.org`) specification. A Foundation library API only contains prototypes without code. Foundation library implementations are provided by MicroEJ Platforms. From a MicroEJ Classpath, Foundation library APIs dependencies are automatically mapped to the associated implementations provided by the platform on which the application is being executed.

A MicroEJ Add-On Library is a MicroEJ library that is implemented on top of MicroEJ Foundation Libraries (100% full Java code). A MicroEJ Add-on Library is distributed in a single JAR file, with most likely a 3 digits version and provides its associated source code.

Foundation and add-on libraries are added to MicroEJ Classpath by the application developer using Ivy (see Section 3.5, “Library Dependency Manager”).

Figure 3.9. MicroEJ Foundation and Add-On Libraries



3.5. Library Dependency Manager

MicroEJ uses Ivy (`http://ant.apache.org/ivy`) as its dependency manager for building MicroEJ classpath.

An Ivy configuration file must be provided in each MicroEJ project to solve classpath dependencies. Multiple Ivy configuration file templates are available depending on the kind of MicroEJ application created.

Example 3.5. Ivy File Template

```
<ivy-module version="2.0">
   <info module=""/>
   <dependencies>
    <!-- Declare a Foundation Library API dependency -->
    <dependency org="ej.api" name="edc" rev="1.2.+"/>
    <dependency org="ej.api" name="bon" rev="1.2.+"/>

    <!-- Declare an Add-On Library dependency -->
    <dependency org="org.kxml2" name="kxml2" rev="2.3.1"/>

    <!-- Declare a test only library dependency -->
    <dependency org="ej.library.test" name="junit" rev="[1.0.0-
RC0,2.0.0-RC0[" conf="test->*" />
   </dependencies>
  </ivy-module>
```

Dependencies are declared within the `<dependencies>` tag

- Foundation libraries are declared using the `"ej.api"` organization. Without this, they will be considered as a regular Add-On libraries and will not be mapped to the associated implementation provided by the platform.
- Add-On libraries are declared with the default runtime configuration. All their declared dependencies will be fetched transitively.

3.6. Central Repository

The MicroEJ Central Repository is the Ivy repository maintained by MicroEJ. It contains Foundation library APIs and numerous Add-On Libraries. Foundation libraries APIs are distributed under the organization `ej.api`. All other artifacts are Add-On libraries.

For more information, please visit `https://developer.microej.com`.

Chapter 4. Additional Tools

4.1. Testsuite with JUnit

MicroEJ allows to run unit tests using the standard JUnit API during the build process of a MicroEJ library or a MicroEJ application. The MicroEJ testsuite engine runs tests on a target Platform and outputs a JUnit XML report.

4.1.1. Principle

JUnit testing can be enabled when using the `microej-javalib` (MicroEJ add-on library) or the `microej-application` (MicroEJ applications) build type. JUnit test cases processing is automatically enabled when the following dependency is declared in the `module.ivy` file of the project.

```
<dependency conf="test->*" org="ej.library.test" name="junit"
 rev="[1.0.0-RC0,2.0.0-RC0["/>
```

When a new JUnit test case class is created in the `src/test/java` folder, a JUnit processor generates MicroEJ compliant classes into a specific source folder named `src-adpgenerated/junit/java`. These files are automatically managed and must not be edited manually.

4.1.2. JUnit Compliance

MicroEJ is compliant with a subset of JUnit version 4. MicroEJ JUnit processor supports the following annotations: `@After`, `@AfterClass`, `@Before`, `@BeforeClass`, `@Ignore`, `@Test`.

Each test case entry point must be declared using the `org.junit.Test` annotation (`@Test` before a method declaration). Please refer to JUnit documentation to get details on usage of other annotations.

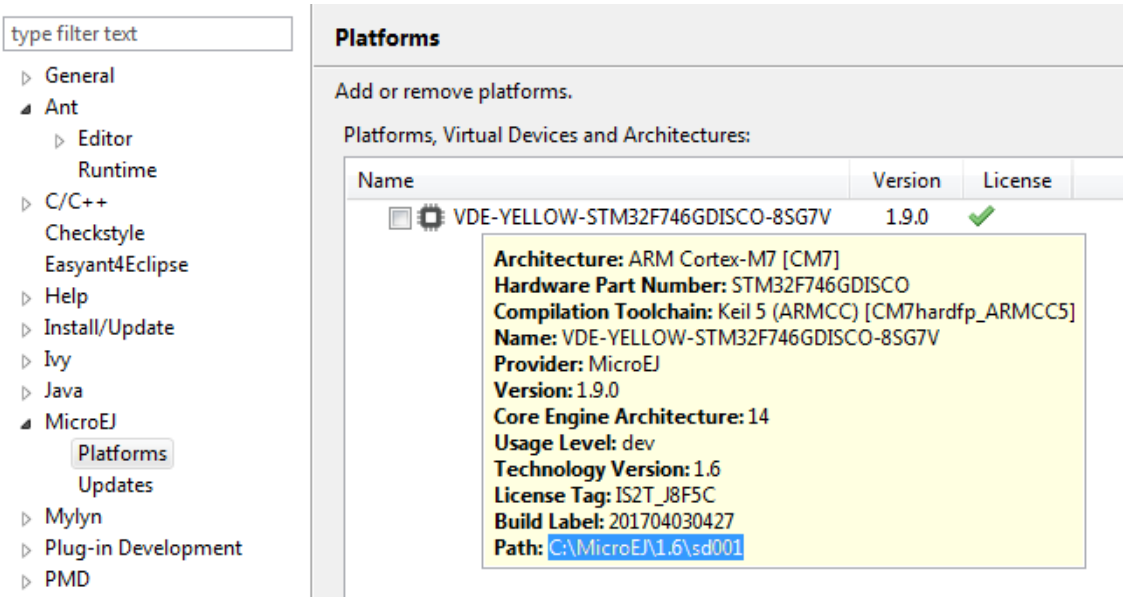
4.1.3. Setup a Platform for Tests

Before running tests, a target platform must be configured in the MicroEJ workspace. The following steps assume that a platform has been previously imported into the MicroEJ Platform repository.

Go to `Window > Preferences > MicroEJ > Platforms` and select the desired platform on which to run the tests.

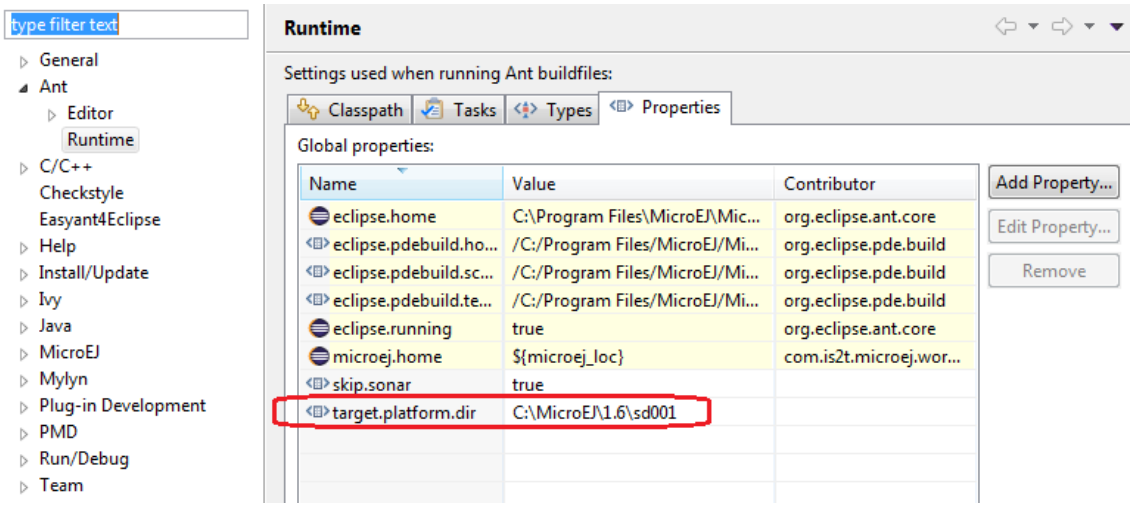
Press `F2` to expand the details.

Select the the platfom path and copy it to the clipboard.



Go to `Window > Preferences > Ant > Runtime` and select the `Properties` tab.

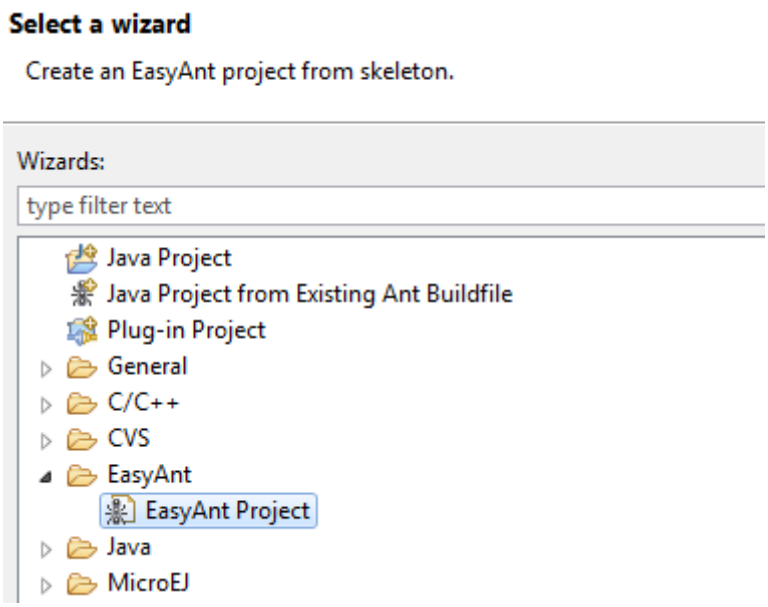
Click on `Add Property...` button and set a new property named `target.platform.dir` with the platform path pasted from the clipboard.



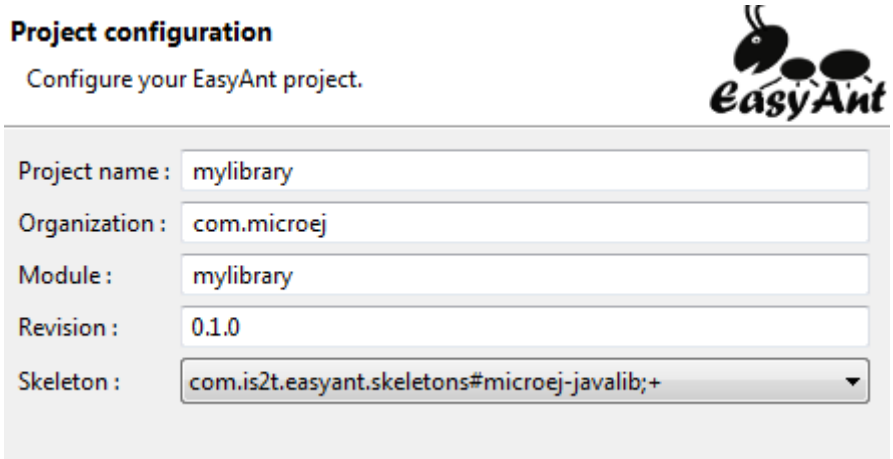
4.1.4. Setup a Project with a JUnit Test Case

This section describes how to create a new JUnit Test Case starting from a new MicroEJ library project.

Select `File > New > Project... > EasyAnt > EasyAnt Project`.



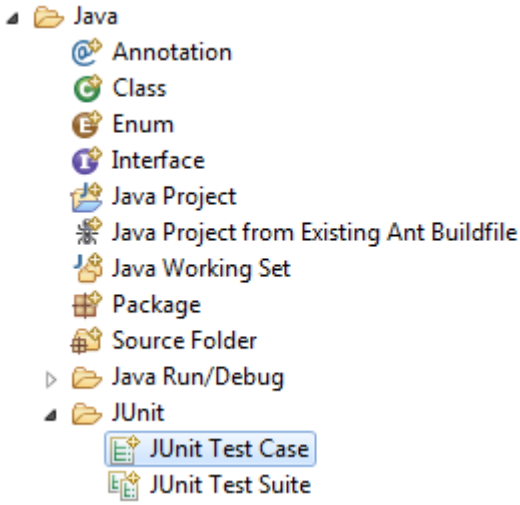
Press `Next`. Fill out project settings and select the `microej-javalib` skeleton



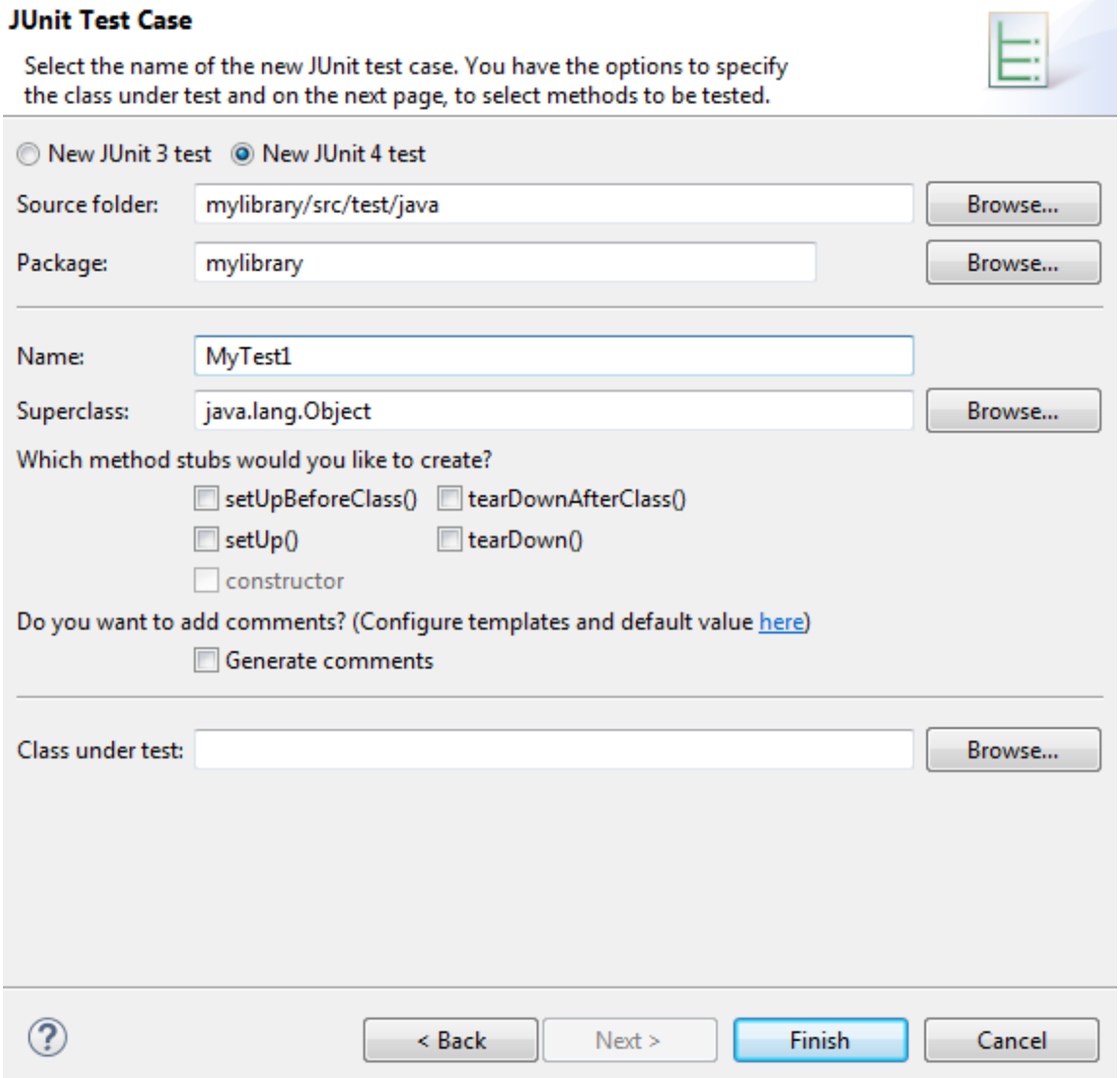
A new project named `mylibrary` is created in the workspace.

Right-click on the `src/test/java` folder and select `New > Other...` menu item.

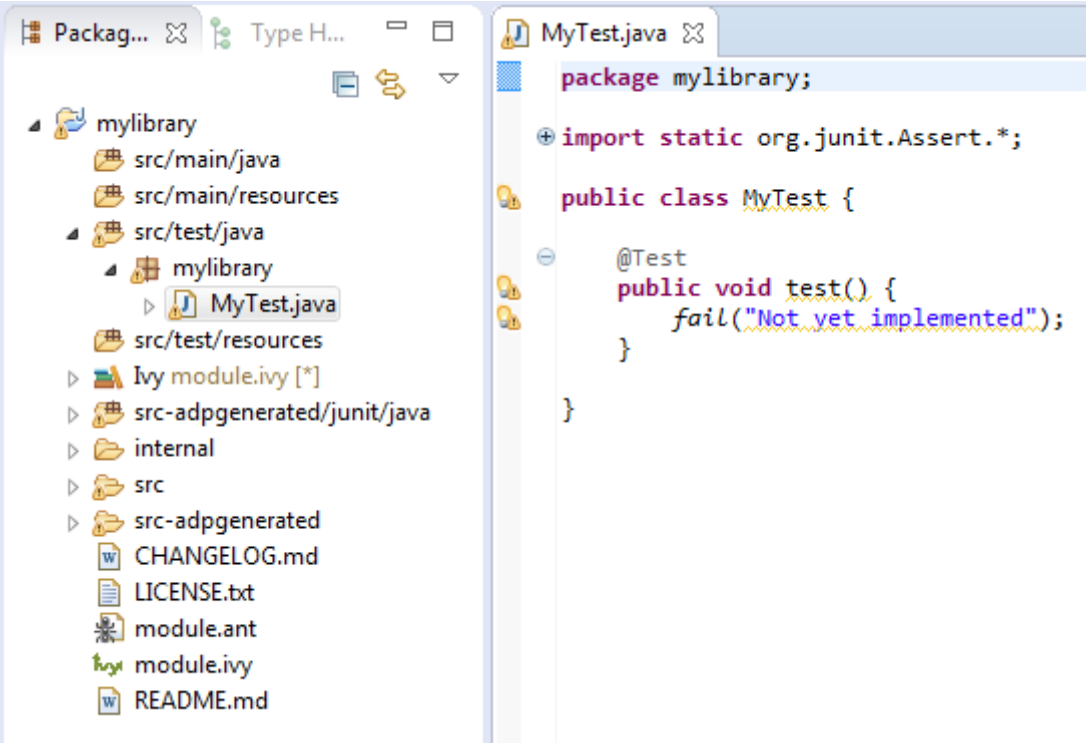
Select the `Java > JUnit > New JUnit Test Case` wizard.



Enter a test name and press `Finish`.



A new JUnit test case class is created with a default failing test case.



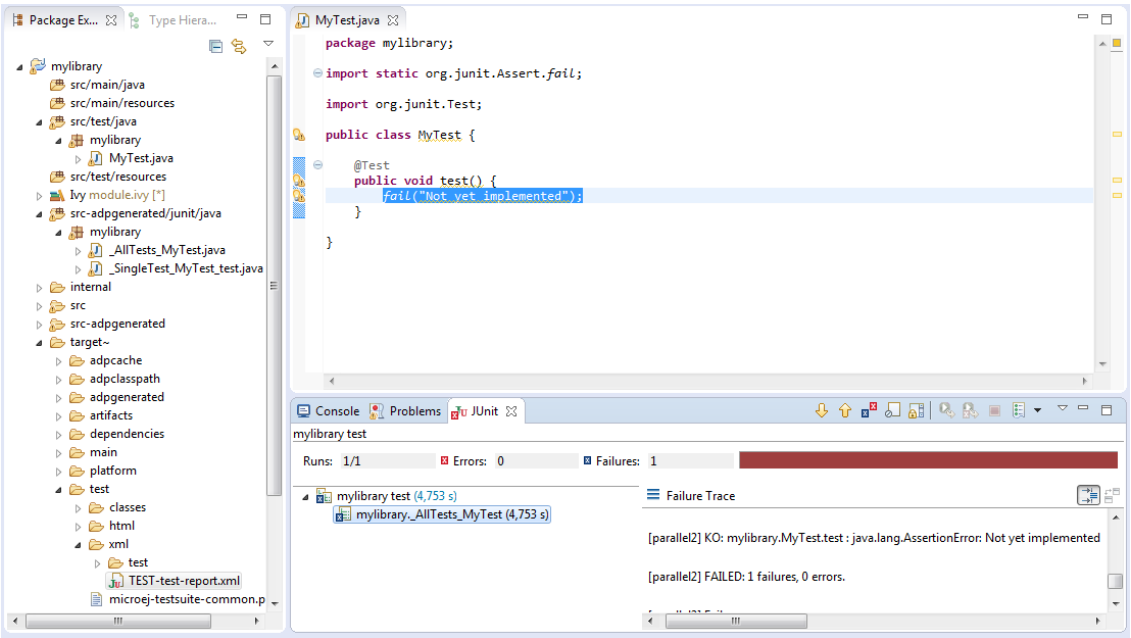
4.1.5. Build and Run a JUnit Testsuite

Right-click on the `mylibrary` project and select `Build with EasyAnt`. After the library is built, the testsuite engine launches available test cases and the build process fails in the console view.

On the `mylibrary` project, right-click and select `Refresh`.

A `target~` folder appears with intermediate build files. The JUnit report is available at `target~\test\xml\TEST-test-report.xml`.

Double-click on the file to open the JUnit testsuite report.



Modify the test case by replacing

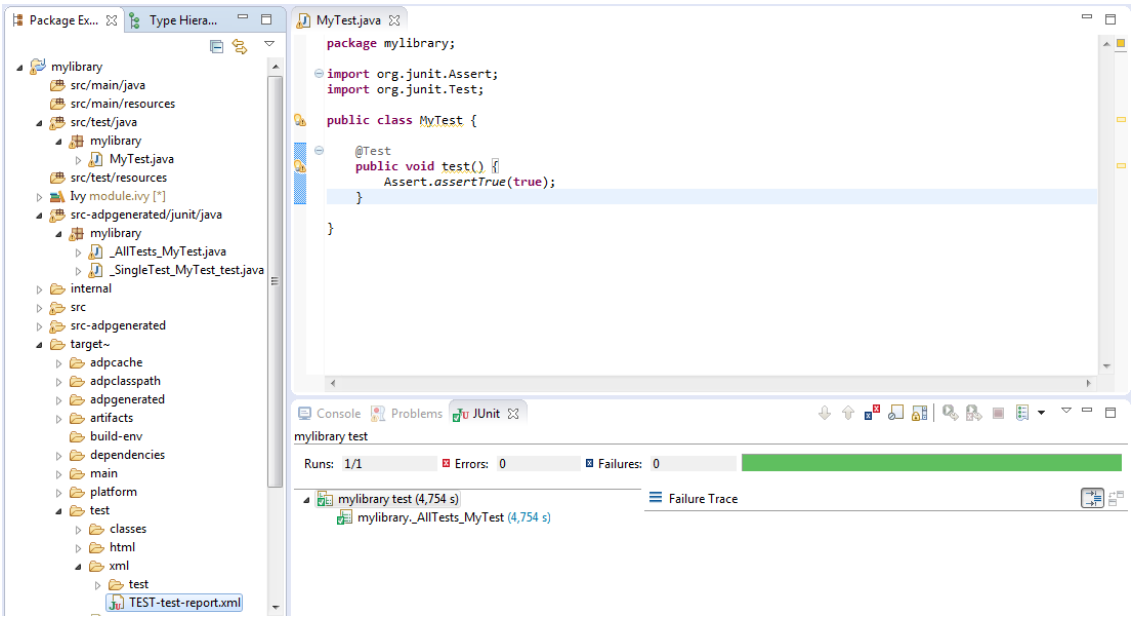
```
fail("Not yet implemented");
```

with

```
Assert.assertTrue(true);
```

Right-click again on the `mylibrary` project and select `Build with EasyAnt`. The test is now successfully executed on the target platform so the MicroEJ add-on library is fully built and published without errors.

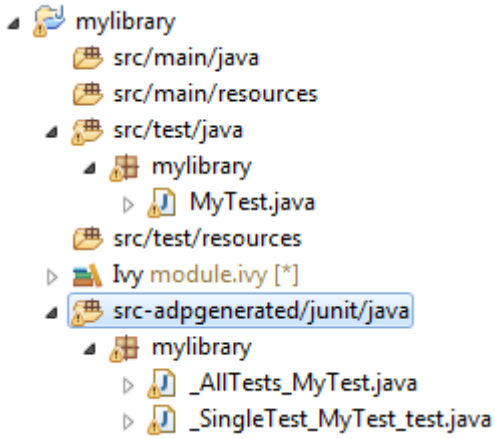
Double-click on the JUnit testsuite report to see the test has been successfully executed.



4.1.6. Advanced Configurations

4.1.6.1. Autogenerated Test Classes

The JUnit processor generates test classes into the `src-adpgenerated/junit/java` folder.



This folder contains:

- `_AllTests_[TestCase].java` files: for each JUnit test case class, a class with a main entry point that sequentially calls all declared test methods.
- `_SingleTest_[TestCase]_[TestMethod].java` files: for each test method of each JUnit test case class, a class with a main entry point that calls the test method.

4.1.6.2. JUnit Test Case to MicroEJ Test Case

The MicroEJ testsuite engine allows to select the classes that will be executed, by setting the following property in the project `module.ivy` file.

```
<ea:property name="test.run.includes.pattern" value="[MicroEJ Test
 Case Include Pattern]"/>
```

The following line consider all JUnit test methods of the same class as a single MicroEJ test case (default behaviour). If at least one JUnit test method fails, the whole test case fails in the JUnit report.

```
<ea:property name="test.run.includes.pattern" value="**/
_AllTests_*.class"/>
```

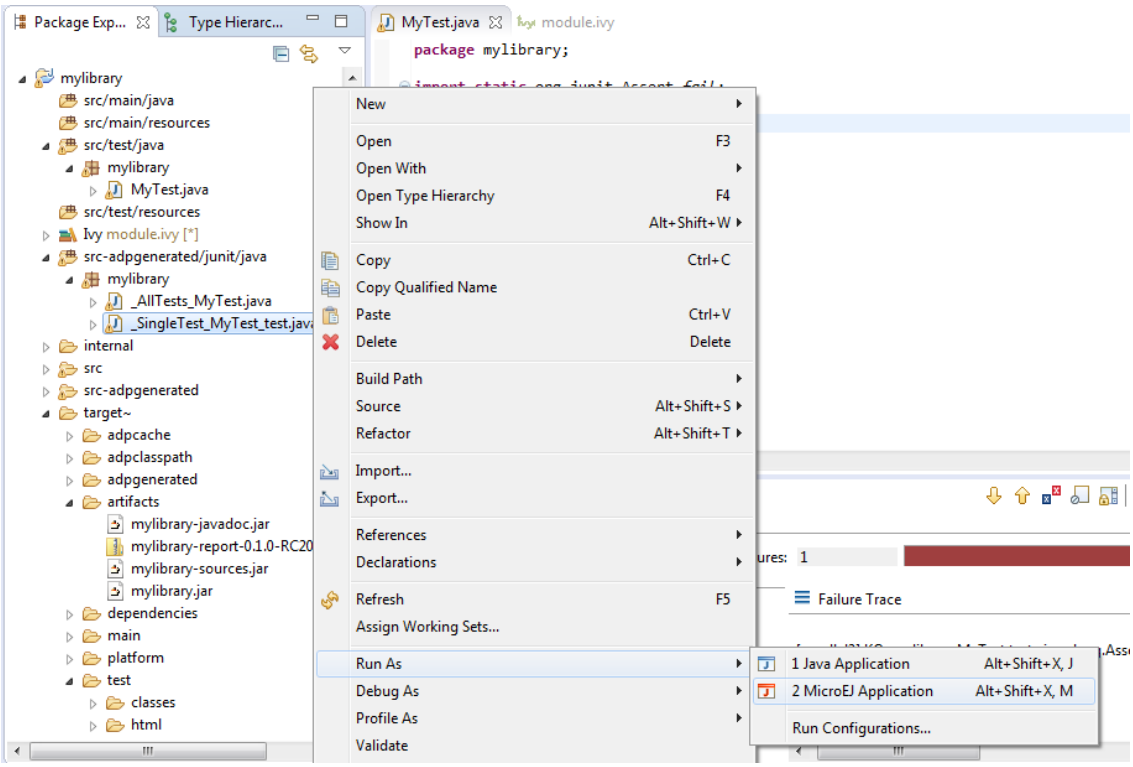
The following line consider each JUnit test method as a dedicated MicroEJ test case. Each test method is viewed independently in the JUnit report, but this may slow down the testsuite execution because a new deployment is done for each test method.

```
<ea:property name="test.run.includes.pattern" value="**/
_SingleTest_*.class"/>
```

4.1.6.3. Run a Single Test Manually

Each test can be run independently as each class contains a main entry point.

In the `src-adpgenerated/junit/java` folder, right-click on the desired autogenerated class (`_SingleTest_[TestCase]_[TestMethod].java`) and select `Run As > MicroEJ Application`.

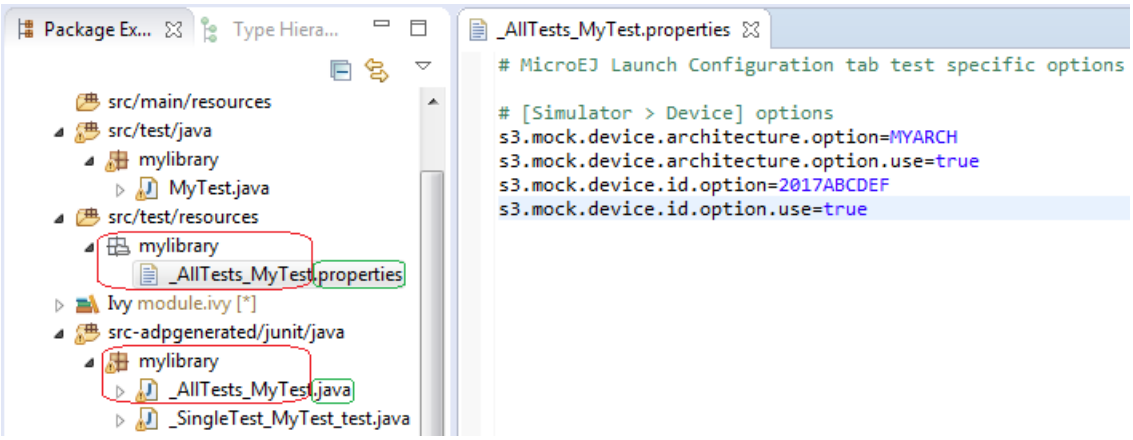


The test is executed on the selected Platform and the output result is dumped into the console.

4.1.6.4. Test Specific Options

The MicroEJ testsuite engine allows to define MicroEJ Launch options specific to each test case. This can be done by defining a file with the same name of the generated test case file with the `.properties` extension instead of the `.java` extension. The file must be put in the `src/test/resources` folder and within the same package than the test case file.

Consult the Application Launch Options Appendix of the Device Developer's Guide to get the list of available options properties.



4.2. Font Designer

MicroEJ Font Designer allows to create embedded fonts files (see Section 3.3.7, “Fonts”) from standard font files formats. The Font Designer documentation is available at: `Help > Help Contents > Font Designer User Guide`.

4.3. Stack Trace Reader

When an application is deployed on a device, stack traces dumped on standard output are not directly readable: non required types (see Section 3.3.2, “Types”) names, methods names and methods line numbers may not have been embedded to save code space. A stack trace dumped on the standard output can be decoded using the Stack Trace Reader tool.

Starting from the MicroEJ application example (see Section 2.3, “Build and Run an Application”), write a new line to dump the currently executed stack trace on the standard output.

Figure 4.1. Code to Dump a Stack Trace

```
module.ivy    Test.java
1  package com.mycompany;
2
3  public class Test {
4
5      public static void main(String[] args) {
6          System.out.println("hello world!");
7          new Exception().printStackTrace();
8      }
9
10 }
```

On successful deployment, the application is started on the device and the following trace is dumped on standard output.

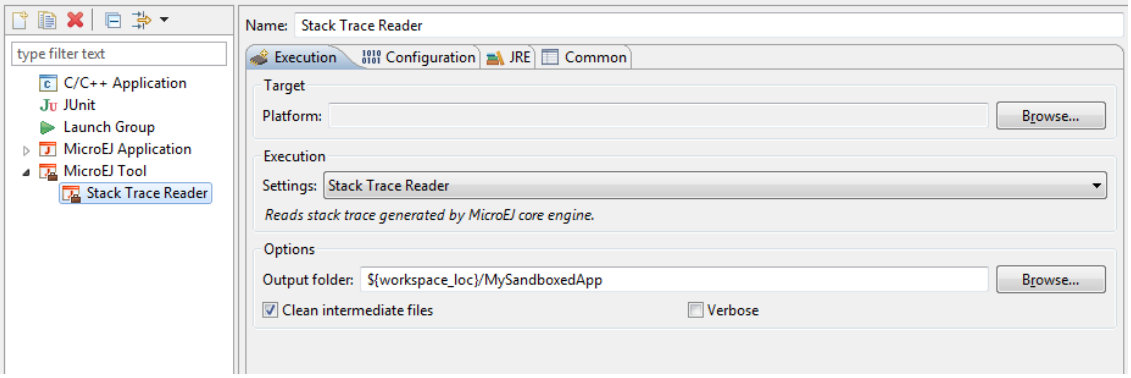
Figure 4.2. Stack Trace Output

```
hello world!
Exception in thread "main" java.lang.Exception
     at java.lang.System.@M:0x800a054:0x800a064@
     at java.lang.Throwable.@M:0x800b6e0:0x800b6f6@
     at java.lang.Throwable.@M:0x800ca7c:0x800caa0@
     at java.lang.Exception.@M:0x800c9e0:0x800c9f0@
     at com.mycompany.Test.@M:0x800b5e0:0x800b608@
     at java.lang.MainThread.@M:0x800c920:0x800c936@
     at java.lang.Thread.@M:0x800ed58:0x800ed64@
     at java.lang.Thread.@M:0x800edc8:0x800edd3@
VM END (exit code = 0)
```

To create a new MicroEJ Tool configuration, right-click on the application project and click on `Run As... > Run Configurations...`.

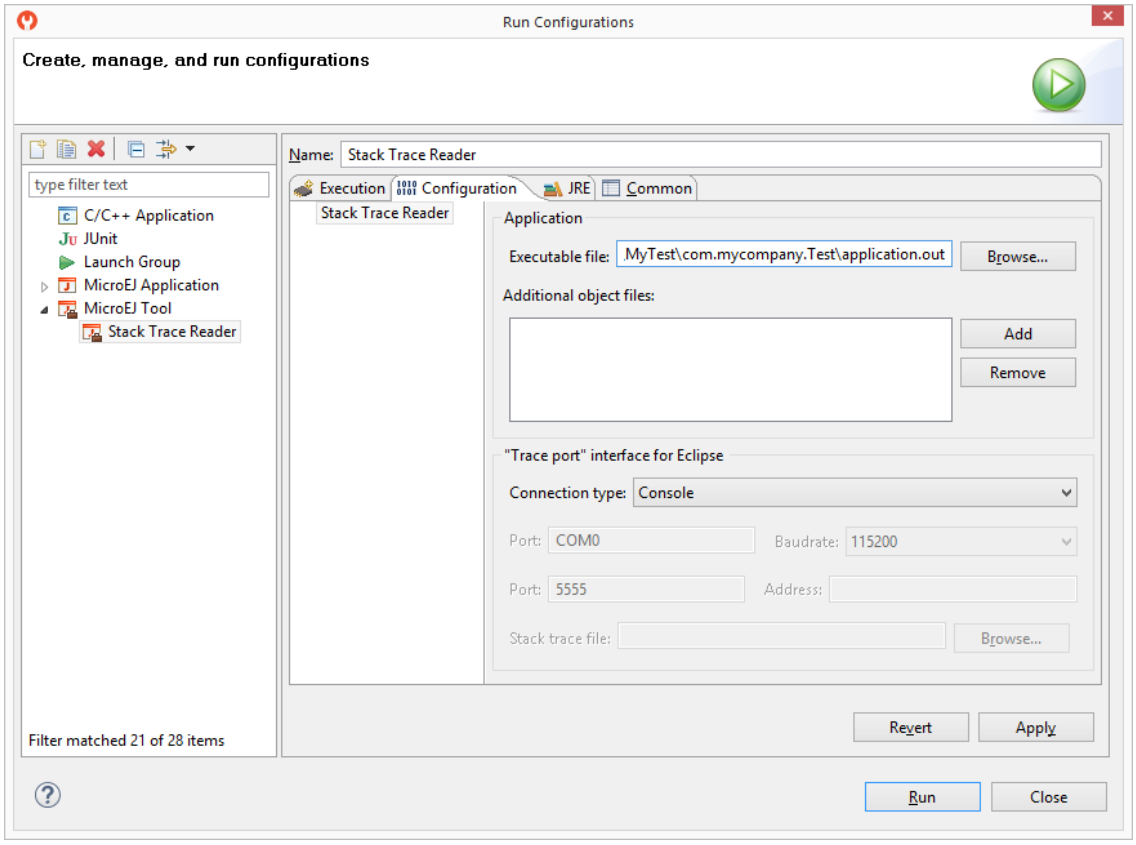
In `Execution` tab, select the `Stack Trace Reader` tool.

Figure 4.3. Select Stack Trace Reader Tool



In `Configuration` tab, browse the previously generated application binary file with debug information (`application.out`)

Figure 4.4. Stack Trace Reader Tool Configuration



Click on Run button and copy/paste the trace into the Eclipse console. The decoded trace is dumped and the line corresponding to the application hook is now readable.

Figure 4.5. Read the Stack Trace

```
Console  Problems  Progress  Front Panel Preview  History  Search  Call Hierarchy
Stack Trace Reader_ [MicroEJ Tool] C:\Program Files\Java\jre1.8.0_66\bin\javaw.exe (Jun 21, 2016, 4:18:16 PM)
=============== [ MicroEJ OS Core Engine Trace ] ===============
[INFO] Paste the MicroEJ OS core engine stack trace here.
Exception in thread "main" java.lang.Exception
     at java.lang.System.@M:0x800a054:0x800a064@
     at java.lang.Throwable.@M:0x800b6e0:0x800b6f6@
     at java.lang.Throwable.@M:0x800ca7c:0x800caa0@
     at java.lang.Exception.@M:0x800c9e0:0x800c9f0@
     at com.mycompany.Test.@M:0x800b5e0:0x800b608@
     at java.lang.MainThread.@M:0x800c920:0x800c936@
     at java.lang.Thread.@M:0x800ed58:0x800ed64@
     at java.lang.Thread.@M:0x800edc8:0x800edd3@Exception in thread "main" java.lang.Exception
     at java.lang.System.getStackTrace(Unknown Source)
     at java.lang.Throwable.fillInStackTrace(Throwable.java:79)
     at java.lang.Throwable.<init>(Throwable.java:25)
     at java.lang.Exception.<init>(Exception.java:6)
     at com.mycompany.Test.main(Test.java:7)
     at java.lang.MainThread.run(Thread.java:770)
     at java.lang.Thread.runWrapper(Thread.java:448)
```

The stack trace reader can simultaneously decode heterogeneous stack traces with lines owned by different applications and the firmware. Other debug information files can be appended using the `Additional object files` option. Lines owned by the firmware can be decoded with the firwmare debug information file (optionally made available by your firmware provider).