



Process and Service Programming

2.2 Process management in Java - ProcessBuilder and Process



I.E.S.
Doctor Balmis

PSP class notes (https://psp2dam.github.io/psp_sources) by Vicente Martínez is licensed under
CC BY-NC-SA 4.0  (<http://creativecommons.org/licenses/by-nc-sa/4.0/?ref=chooser-v1>)

2.2 Process management in Java ProcessBuilder and Process

- 2.2.1 Preparation and setting of a process
 - Setting the command at runtime
 - Additional settings for a process
- 2.2.2 Process control from parent
- 2.2.3 Spawn a java application from a class into the same project
- 2.2.4 Current Java Process Information
 - Getting information about the current process
 - Getting information about a child process from parent

2.2.1 Preparation and setting of a process

The class to set the running attributes for a new process, before it is being run, is the `ProcessBuilder` class.

Specification `java.lang.ProcessBuilder`

(<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/ProcessBuilder.html>)

This is an auxiliary class for the `Process` and is instantiated to manage a collection of process attributes. We can invoke the `start` method to create a new process with the attributes defined by the instance of the `ProcessBuilder` class.

Repeated calls to the `start` method would create a new process with the same attributes.

The `ProcessBuilder` class defines two constructors, such as:

```
ProcessBuilder(List<String> command)
ProcessBuilder(String... command)
```

java

The meaning implied by the parameters passed to both constructors is same. In the first constructor, the command to be executed, along with command line arguments, is passed in a list of strings. And, in the second constructor, the command and the command line arguments are specified through the varargs parameter. We can use either of the constructors, depending upon the way to pass the parameter.



Arguments vs Parameters

If we want to launch a command with parameters, the command cannot be sent to `ProcessBuilder` in raw mode, it must be processed and converted into a `List` in order to make it work.

```
1 // Different modes to pass the command to ProcessBuilder constructors
2 // 1st mode: using a string. It fails with parameters,
3 // Only works with commands having arguments
4 String command1 = "notepad.exe prueba1.txt"
5 ProcessBuilder pb = new ProcessBuilder(command1);
6
7 // 2nd mode: using an array of strings. It also works with parameters
8 String[] command2 = {"cmd", "/c", "dir", "/o"};
9 ProcessBuilder pb = new ProcessBuilder(command2);
10
11 // 3rd mode: using a string and splitting it to convert into a List
12 String command3 = "c:/windows/system32/shutdown -s -t 0";
13 // Regular expression \s means splitting the string by blank spaces
```

java

```
14 ProcessBuilder pb = new ProcessBuilder(command3.split("\\s"));
```

! OS shutdown

You can use shutdown -s command to shutdown system. For windows OS, you need to provide full path of shutdown command e.g. C:\Windows\System32\shutdown.

Here you can use -s switch to shutdown system, -r switch to restart system, -h to put the system into hibernation, and -t switch to specify time delay.

Windows shutdown reference (<https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/shutdown>)

? Activity psp.activities.U2A1_Shutdowner

Create a new Java application project (package psp.activities & main class U2A1_Shutdowner) Using the command line, ask the user for the action he wants to do with the computer (shutdown ,restart or suspend) and how much time he needs before shutting down the system.

Find information about the shutdown command in GNU/Linux and make your app work in both systems.

Your app has to prepare the right command for the answers the user has given and for the OS it is running on.

Get the ProcessBuilder.command() result and show it on the console in a readable format.

Setting the command at runtime

If we want to set the command to be run at runtime, or at the time the ProcessBuilder instance is created we still don't know the command, it can be set later by using the command(String).

The same way as the constructors, we have two versions of command method

```
command(List<String> command)
command(String... command)
```

java

and there's also another command method, without parameters, to retrieve the command and parameters already set for the ProcessBuilder instance. Once we have the parameters list, we can modify it using List methods.

```
1 // Sets and modifies the command after ProcessBuilder object is created
2 String command = "java -jar install.jar -install"; // tmp dir is missing
3 ProcessBuilder pbuilder = new ProcessBuilder(command.split("\\s"));
4 if (isWindows) {
5     pbuilder.command().add(0, "cmd"); // Sets the 1st element
6     pbuilder.command().add(1, "/c"); // Sets the 2nd element
7     pbuilder.command().add("c:/temp"); // Sets the last element
8     // Command to run cmd /c java -jar install.jar -install c:/temp
9 } else {
10     pbuilder.command().add(0, "sh"); // Sets the 1st element
11     pbuilder.command().add(1, "-c"); // Sets the 2nd element
12     pbuilder.command().add("/tmp"); // Sets the last element
13     // Command to run: sh -c java -jar install.jar -install /tmp
14 }
```

java

```
15
16 // Starts the process
17 pbuilder.start();
```

Additional settings for a process

Some of the settings that can be changed for a process are:

- Set the working directory where the process will be run We can override the default working directory of the current process by calling the directory method and passing a File object. **By default, the current working directory is set to the value returned by the user.dir system property.**

```
1 // Change working directory for the running process
2 pbuilder.directory(new File(System.getProperty("user.home")));
```

java

- Set-up a custom key-value map and modify an existing one using builder.environment()

```
1 // Retrieve and modify the process environment
2 Map<String, String> environment = pbuilder.environment();
3 // Get the PATH environment variable and add a new directory
4 String systemPath = environment.get("path") + ";c:/users/public";
5 environment.replace("path", systemPath);
6 // Add a new environment variable and use it as a part of the command
7 environment.put("GREETING", "Hola Mundo");
8 processBuilder.command("/bin/bash", "-c", "echo $GREETING");
```

java



Environment variables vs System properties

With Runtime we also accessed System properties, that are different from this environment ones.

- Redirect input and output streams to custom replacements
- Inherit both of them to the streams of the current JVM process using builder.inheritIO()

This two settings will be covered later in this unit.



Activity psp.activities.U2A2_WorkingDirectory

Create a new Java application project (package psp.activities & main class U2A2_WorkingDirectory) Prepare a process to run the dir/lis command to check that the directory listing is for the directory pointed by the user.dir property. In the same application, change the value for the user.dir property. Finally, set a working directory for the process.

Print the user.dir environment value for the three scenarios after being changed.

2.2.2 Process control from parent

The `Process` is an abstract class defined in the `java.lang` package that encapsulates the runtime information of a program in execution. The `start` method invoked by the `ProcessBuilder` class returns a reference to this class instance. There is another way to create an instance of this class, through the `exec` method of the `Runtime` instance.

The methods defined by the `Process` class can be used to perform input/output operations from the process, check the exit status of the process, wait for it to complete, and terminate the process. These methods, however, are not built to work on special processes of the native platform like daemon processes, shell scripts, and so on.

Specification `java.lang.Process` (<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Process.html>)

! Input/ output from the child process

Intriguingly, the process created by the `start()` method does not own a console. Instead, it redirects (stdin, stdout, stderr) to the parent process. If need be, we can access them via streams obtained using methods defined in the class, such as `getInputStream()`, `getOutputStream()` and `getErrorStream()`. These are the ways we can feed input to and get results from the sub processes.

Some of the common methods defined in this class are:

method	Description
<code>int exitValue()</code>	Exit code returned from the process executed
<code>Boolean isAlive()</code>	Checks if the invoking process is still running.
<code>int waitFor()</code>	Parent process waits for the child process to end. The integer value returned by the method is the exit code by the process.
<code>Boolean waitFor(long timeout, TimeUnit unit)</code>	Overloaded method of previous one. We can specify the wait time. This method returns true if the process has terminated and false if timeout has occurred.
<code>void destroy()</code>	These two methods are used to kill or terminate the process. One, the second, just does it forcibly.
<code>Process destroyForcibly()</code>	

Let's write a simple Java program to open an application as a separate process. After it is opened, the program would wait for, say, 10 seconds and then destroy the process, which will immediately close the application.

```
1 public class ProcessDemo {
2
3     public static void main(String[] args) throws Exception {
4
5         ProcessBuilder pb = new ProcessBuilder("C:/Program Files (x86)/Notepad+/notepad++.exe");
6         // Effectively launch the process
7         Process p = pb.start();
8         // Check if process is alive or not
9         boolean alive = p.isAlive();
10        // Wait for the process to end for 10 seconds.
11        if (p.waitFor(10, TimeUnit.SECONDS)) {
12            System.out.println("Process has finished");
13        } else {
14            System.out.println("Timeout. Process hasn't finished");
15        }
16    }
17 }
```

java

```
15     }
16     // Force process termination.
17     p.destroy();
18     // Check again if process remains alive
19     alive = p.isAlive();
20     // Get the process exit value
21     int status = p.exitValue();
22 }
23 }
```



Exit codes

An exit code, or sometimes known as a return code, is the code returned to a parent process by an executable. The standard exit code is 0 for success and any number from 1 to 255 for anything else.



Activity psp.activities.U2A3_ExitValue

Create a new Java application project (package psp.activities & main class U2A3_ExitValue) Prepare a process to run different commands (notepad, calc, shell commands) one after each other, and make your application get their exit code. Print it.

Commands can be hardcoded. As an **optional** improvement for this activity you can ask the user for the command and make your app interactive. There must be an option to exit the app (empty command for instance).

Try with non-existing applications or using wrong arguments/parameters for commands.

Can you force a process not to be successful?

How can you know your own process exit code?



Exceptions management

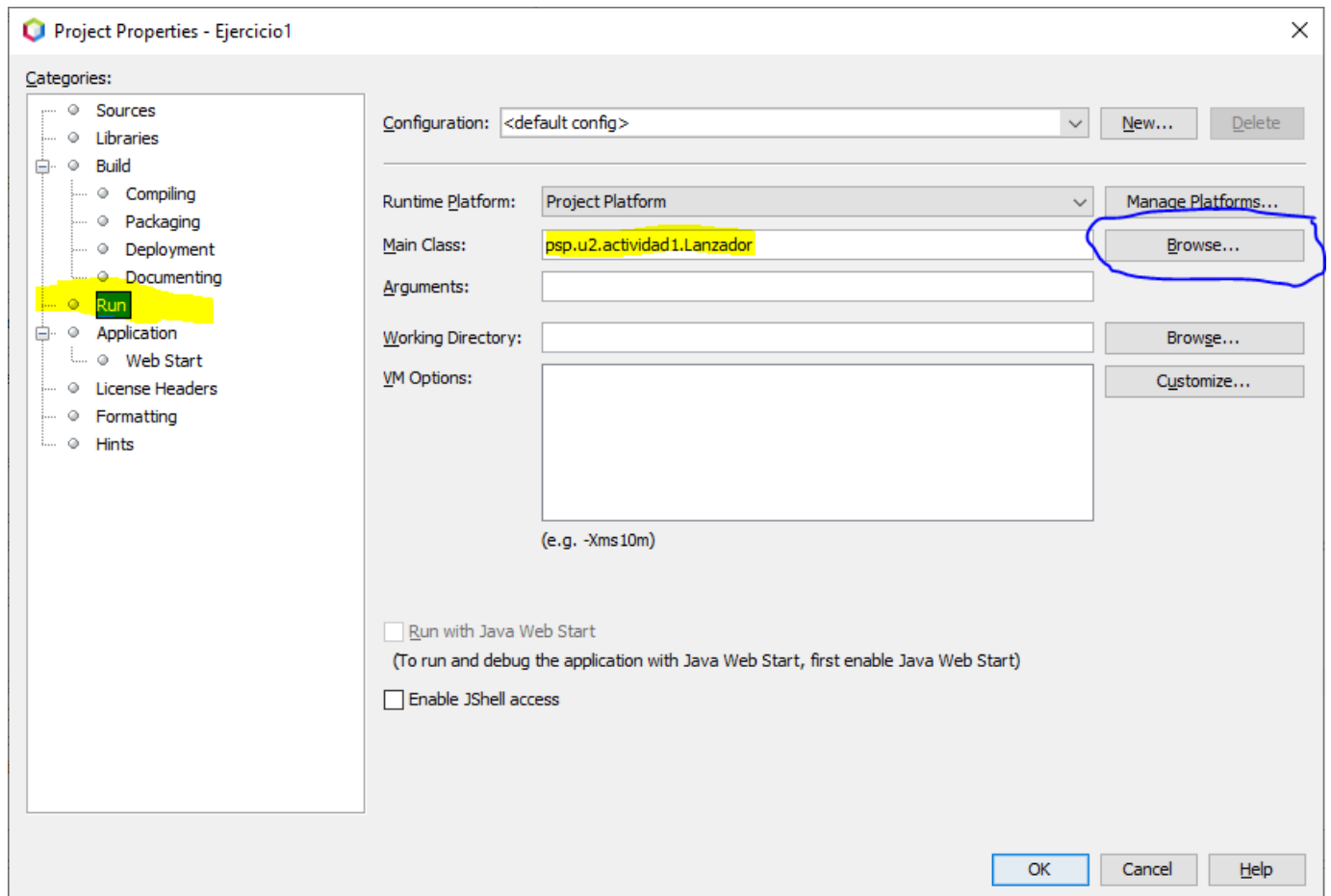
Call to method **waitFor** implies that the parent process gets locked until child process ends, or until a signal from the system (Exception) is received.

It's better to handle exceptions than to throw them to upper levels.

2.2.3 Spawn a java application from a class into the same project

For some activities you'll be required to create the parent and the child processes as Java applications. Then, from one the classes you will need to launch the other one.

This implies that both classes are gonna have a main method. So, in the project properties we'll need to set which is the main class that will be run first, usually the `Launcher` class (parent process).



Before one class can run the other, at least the second one (child process) must be compiled, that is, the .class file has to be generated into the build/classes directory.

```
C:.\
├── build
│   ├── classes
│   │   ├── psp
│   │   │   ├── u2
│   │   │   │   ├── actividad1
│   │   │   │   │   └── lanzador
│   │   └── empty
│   ├── generated-sources
│   │   └── ap-source-output
│   └── test
├── dist
├── nbproject
│   └── private
├── src
│   ├── psp
│   │   ├── u2
│   │   │   └── actividad1
│   └── test
└── test
```

Then and only then we can set the process environment to spawn a new process from an existing class. Here is the sample code

```
1 // Prepare the environment and the command
2 ProcessBuilder pb = new ProcessBuilder("java", "psp.u2.actividad10.Sumador");
3 pb.directory(new File("build/classes"));
4 Process p = pb.start();
```

java



Activity psp.activities.U2A4_Launcher

Create a new Java application project (package psp.activities & main class U2A4_Launcher).

Into the project create another class, U2A4_Commander with a main method that receives a program name as a unique parameter in the main. Make this application to create and run a process for the program and wait until that process has finished.

This class will return always the same value the launched program did.

System.exit() method

- Zero. The zero status code should be used when the program execution went fine, i.e., the program is terminated successfully.
- Non-Zero. A nonzero status code indicates abnormal termination. Java allows us to use different values for different kinds of errors.

Now, make the U2A4_Launcher class ask the user for an application name and launch the Commander class passing it the name of the application entered by the user.

Get the exitValue from Commander and show it's value, telling if the process worked fine or if it failed.



Child classes programming

Every class must be coded to be run independently or just as a child process. Think that in Netbeans all classes are run as child classes from the IDE.

That's why the code, child or parent, needs to be done without thinking how they are gonna be called. The code must be **independent** just like processes are one from each other.

2.2.4 Current Java Process Information

We can now obtain a lot of information about the process via the API `java.lang.ProcessHandle.Info` API:

- the command used to start the process
- the arguments of the command
- time instant when the process was started
- total time spent by it and the user who created it

Getting information about the current process

Here's how we can do that for the current process:

```
1 // Get information about the current process
2 ProcessHandle processHandle = ProcessHandle.current();
3 ProcessHandle.Info processInfo = processHandle.info();
4
5 System.out.println("PID: " + processHandle.pid());
6 System.out.println("Arguments: " + processInfo.arguments());
7 System.out.println("Command: " + processInfo.command());
8 System.out.println("Instant: " + processInfo.startInstant());
9 System.out.println("Total CPU duration: " + processInfo.totalCpuDuration());
10 System.out.println("User: " + processInfo.user());
```

java

Getting information about a child process from parent

It is also possible to get the process information of a newly spawned process. In this case, after we spawn the process and get an instance of the `java.lang.Process`, we invoke the `toHandle()` method on it to get an instance of `java.lang.ProcessHandle`.

```
1 // Get information about a child process from parent
2 Process process = processBuilder.inheritIO().start();
3 ProcessHandle childProcessHandle = process.toHandle();
4 ProcessHandle.Info childProcessInfo = childProcessHandle.info();
```

java

The rest of the details remain the same as in the section above