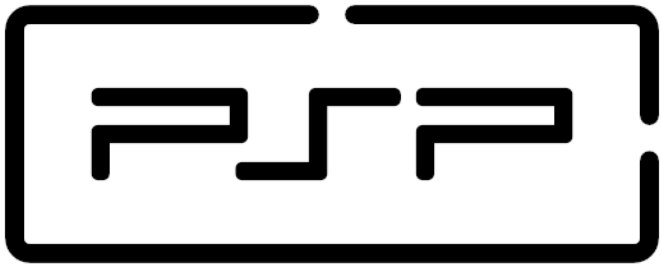

Process and Service Programming

2.2 Process management in Java - ProcessBuilder/Process

I.E.S.
Doctor Balmis

2.2 Process management in Java ProcessBuilder/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

2.2.1 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

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:

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

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.

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.

```java
// Different modes to pass the command to ProcessBuilder constructors
// 1st mode: using a string. It fails with parameters,
// Only works with commands having arguments
String command1 = "notepad.exe prueba1.txt"
ProcessBuilder pb = new ProcessBuilder(command1);

// 2nd mode: using an array of strings. It also works with parameters
String[] command2 = {"cmd", "/c", "dir", "/o"};
ProcessBuilder pb = new ProcessBuilder(command2);

// 3rd mode: using a string and splitting it to convert into a List
String command3 =  "c:/windows/system32/shutdown -s -t 0";
// Regular expresion \s means splitting the string by blank spaces
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

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

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

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.

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

// Starts the process
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**.

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

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

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

> **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 tu run the dir/ls 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 an 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

> **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 getErrorSteam(). 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
int exitValue()	Exit code returned from the process executed
Boolean isAlive()	Checks if the invoking process is still running.
int waitFor()	Parent process waits for the child process to end. The integer value returned by the method is the exit code by the process.
Boolean waitFor(long timeOut, TimeUnit unit)	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.
void destroy()	These two methods are used to kill or terminate the process. One, the second, just does it forcibly.

method	Description
Process destroyForcibly()	

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.

```java
public class ProcessDemo {

   public static void main(String[] args) throws Exception {

      ProcessBuilder pb = new ProcessBuilder("firefox");
      // Effectively launch the process
      Process p = pb.start();
      // Check is process is alive or not
      boolean alive = p.isAlive();
      // Wait for the process to end for 10 seconds.
      if (p.waitFor(10, TimeUnit.SECONDS)) {
          System.out.println("Process has finished");
      } else {
          System.out.println("Timeout. Process hasn't finished");
      }
      // Force process termination.
      p.destroy();
      // Check again if process remains alive
      boolean alive = p.isAlive();
      // Get the process exit value
      int status = p.exitValue();
   }
}
```

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 with wrong arguments 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.