

## CHAPTER 11

# Process API

In this chapter, you will learn:

- What the Process API is
- How to interact with the current process running the Java application
- How to create a native process
- How to get information about a new process
- How to get information about the current process
- How to get information about all system processes
- How to set permissions to create, query, and manage native processes

All example programs in this chapter are members of a `jdojo.process` module, as declared in Listing 11-1.

**Listing 11-1.** The Declaration of a `jdojo.process` Module

```
// module-info.java
module jdojo.process {
    exports com.jdojo.process;
}
```

## What Is the Process API?

The Process API consists of classes and interfaces that let you work with native processes in Java programs. Using the API, you can

- Create new native processes from Java code.
- Get process handles for native processes, whether they were created by Java code or by other means.

- Destroy running native processes.
- Query processes for liveness and their other attributes.
- Get the list of child processes and the parent process of a process.
- Get the process ID (PID) of native processes.
- Get the input, output, and error streams of newly created processes.
- Wait for a process to terminate.
- Execute a task when a process terminates.

The Process API is small. It consists of the classes and interfaces listed in Table 11-1. I explain these classes and interfaces in detail with examples in the following sections.

**Table 11-1.** *Classes and Interfaces for the Process API*

Class/Interface	Description
Runtime	It is a singleton class whose sole instance represents the runtime environment of a Java application.
ProcessBuilder	An instance of the <code>ProcessBuilder</code> class holds a set of attributes for a process. Calling its <code>start()</code> method starts a native process and returns an instance of the <code>Process</code> class that represents the native process. You can call its <code>start()</code> method multiple times; each time, it starts a new process using the attributes held in the <code>ProcessBuilder</code> instance.
<code>ProcessBuilder.Redirect</code>	It is a static nested class that represents a source of process input or a destination of process output.
<code>Process</code>	It is an abstract class whose instances represent native processes started by the current Java program using the <code>start()</code> method of a <code>ProcessBuilder</code> or the <code>exec()</code> method of a <code>Runtime</code> .
<code>ProcessHandle</code>	It is an interface whose instances represent handles to native processes whether they were started by the current Java program or by any other means. You can control and query the state of the native process using this handle.
<code>ProcessHandle.Info</code>	An instance of the <code>ProcessHandle.Info</code> interface represents a snapshot of the attributes of a process.

In Java, you are able to start native processes and work with their input, output, and error streams. Also, it is possible to work with native processes that you did not start and to query the details of processes. For the latter, you use an interface named `ProcessHandle`, from inside the Process API. An instance of the `ProcessHandle` interface identifies a native process; it lets you query the process state and manage the process.

Compare the `Process` class and the `ProcessHandle` interface. An instance of the `Process` class represents a native process started by the current Java program, whereas an instance of the `ProcessHandle` interface represents a native process whether it was started by the current Java program or by other means. The `Process` class contains a `toHandle()` method that returns a `ProcessHandle`.

An instance of the `ProcessHandle.Info` interface represents a snapshot of the attributes of a process. Note that processes are implemented differently by different operating systems, so their attributes vary. The state of a process may change anytime, for example, the CPU time used by the process increases whenever the process gets more CPU time. To get the latest information on a process, you need to use the `info()` method of the `ProcessHandle` interface at the time you need it, which will return a new instance of the `ProcessHandle.Info` interface.

All examples in this chapter were run on Ubuntu Linux. You may get a different output when you run these programs on your machine using Windows or any other different operating system.

---

**Note** The CLI code snippets can easily be converted to their Windows counterpart by adapting the executable and argument file paths.

---

## Knowing the Runtime Environment

Every Java application has an instance of the `Runtime` class that lets you query and interact with the runtime environment in which the current Java application is running. The `Runtime` class is a singleton. You can get its sole instance using the `getRuntime()` static method of this class:

```
// Get the instance of the Runtime
Runtime runtime = Runtime.getRuntime();
```

Using the `Runtime`, you can know the maximum memory that the current JVM can use, the currently allocated memory in the JVM, and the free memory in the JVM. Here are the three methods that let you query the JVM's memory in bytes:

- `long maxMemory()`
- `long totalMemory()`
- `long freeMemory()`

JVM allocates memory lazily. The `maxMemory()` method returns the maximum amount of memory that the JVM can allocate. The method returns `Long.MAX_VALUE` if there is no maximum memory limit.

The `totalMemory()` method returns the currently allocated memory by the JVM out of the maximum memory it can allocate. When the JVM needs more memory, it allocates more memory, and the `totalMemory()` method will return the currently allocated memory. The JVM can allocate maximum memory up to the amount returned by the `maxMemory()` method.

The `freeMemory()` method returns the unused memory out of the currently allocated memory by the JVM. How do you know the memory used by the JVM? The following formula will give you the memory used by the JVM at a specific point in time:

Used Memory = Total Memory - Free Memory

Use the `availableProcessors()` method to get the number of available processors to the JVM.

Use the `version()` method to get a `Runtime.Version` that represents the version of the Java runtime environment. Refer to the Javadoc for the `Runtime.Version` class for more details about the JDK/JRE versioning scheme. Listing 11-2 shows you a few applications of the `Runtime` class in querying the Java runtime environment. You may get a different output.

### **Listing 11-2.** Querying the Java Runtime Environment

```
// QueryingRuntime.java
package com.jdojo.process;
public class QueryingRuntime {
    public static void main(String[] args) {
        // Get the Runtime instance
        Runtime rt = Runtime.getRuntime();
```

```

// Get the JVM memory
long maxMemory = rt.maxMemory();
long totalMemory = rt.totalMemory();
long freeMemory = rt.freeMemory();
long usedMemory = totalMemory - freeMemory;
System.out.format(
    "Max memory = %d, Total memory = %d,"
    + "Free memory = %d, Used memory = %d.%n",
    maxMemory, totalMemory, freeMemory,
    usedMemory);
// Print the number of processors available to
// the JVM
int processors = rt.availableProcessors();
System.out.format("Number of processors = %d%n",
    processors);
// Print the version of the Java runtime
Runtime.Version version = rt.version();
System.out.format("Version = %s%n",
    version);
}
}

Max memory = 3126853632,
    Total memory = 201326592,
    Free memory = 198351728,
    Used memory = 2974864.
Number of processors = 8
Version = 17+01-123

```

You can invoke the garbage collection using the `gc()` method of the `Runtime` class. The `System.gc()` static method is the convenience method for the `Runtime.getRuntime().gc()`.

---

**Note** Method `gc()` is just a hint for the OS to start garbage collection at the next convenient time slot. You must not rely on the garbage collection to start immediately if `gc()` gets called.

---

You can terminate the JVM using the `exit(int status)` method of the `Runtime` class. The `System.exit()` static method is a convenience method for `Runtime.getRuntime().exit()`. By convention, a non-zero value for the status indicates an abnormal termination of the JVM. You can forcibly terminate the JVM using the `halt()` method of the `Runtime` class.

You can add and remove shutdown hooks to the JVM using the `addShutdownHook(Thread hook)` and `removeShutdownHook(Thread hook)` methods of the `Runtime` class. A shutdown hook is a thread, which is initialized, but not started. The JVM starts the thread registered as the shutdown hook when it is terminated.

Use one of its `exec()` overloaded methods to start a native process. You should use the `ProcessBuilder` class to start a native process. The `exec()` method of the `Runtime` class internally uses the `ProcessBuilder` class.

## The Current Process

The `current()` static method of the `ProcessHandle` interface returns the handle of the current process. Note that the current process returned by this method is always the Java process that is executing the code:

```
// Get the handle of the current process
ProcessHandle current = ProcessHandle.current();
```

Once you get the handle of the current process, you can use methods of the `ProcessHandle` interface to get details about the process. Refer to the next section for an example on how to get information about the current process.

---

**Note** You cannot kill the current process. Attempting to kill the current process by using the `destroy()` or `destroyForcibly()` method of the `ProcessHandle` interface results in an `IllegalStateException`.

---

## Querying the Process State

You can use methods in the `ProcessHandle` interface to query the state of a process. Table 11-2 lists this interface's commonly used methods with brief descriptions. Note that many of these methods return the snapshot of the state of a process that was true

when the snapshot was taken. There is no guarantee that the process will still be in the same state when you use its attributes later because processes are created, run, and destroyed asynchronously.

**Table 11-2.** *Methods in the ProcessHandle Interface*

Method	Description
<code>static Stream&lt;ProcessHandle&gt; allProcesses()</code>	Returns a snapshot of all processes in the OS that are visible to the current process.
<code>Stream&lt;ProcessHandle&gt; children()</code>	Returns a snapshot of the current direct children of the process. Use the <code>descendants()</code> method to get a list of children at all levels, for example, child processes, grandchild processes, great grandchild processes, etc.
<code>static ProcessHandle current()</code>	Returns a <code>ProcessHandle</code> for the current process, which is the Java process executing this method call.
<code>Stream&lt;ProcessHandle&gt; descendants()</code>	Returns a snapshot of the descendants of the process. Compare it to the <code>children()</code> method, which returns only direct descendants of the process.
<code>boolean destroy()</code>	Requests the process to be killed. Returns <code>true</code> if termination of the process was successfully requested, <code>false</code> otherwise. Whether you can kill a process depends on operating system access control.
<code>boolean destroyForcibly()</code>	Requests the process to be killed forcibly. Returns <code>true</code> if termination of the process was successfully requested, <code>false</code> otherwise. Killing a process forcibly terminates the process immediately, whereas a normal termination allows a process to shut down cleanly. Whether you can kill a process depends on operating system access control.
<code>ProcessHandle.Info info()</code>	Returns a snapshot of information about the process.

*(continued)*

**Table 11-2.** *(continued)*

Method	Description
<code>boolean isAlive()</code>	Returns <code>true</code> if the process represented by this <code>ProcessHandle</code> has not yet terminated, <code>false</code> otherwise. Note that this method may return <code>true</code> for some time after you have successfully requested to terminate the process because the process will be terminated asynchronously.
<code>static Optional&lt;ProcessHandle&gt; of(long pid)</code>	Returns an <code>Optional&lt;ProcessHandle&gt;</code> for an existing native process. Returns an empty <code>Optional</code> if a process with the specified <code>pid</code> does not exist.
<code>CompletableFuture &lt;ProcessHandle&gt; onExit()</code>	Returns a <code>CompletableFuture &lt;ProcessHandle&gt;</code> for the termination of the process. You can use the returned object to add a task that will be executed when the process terminates. Calling this method on the current process throws an <code>IllegalStateException</code> .
<code>Optional&lt;ProcessHandle&gt; parent()</code>	Returns an <code>Optional&lt;ProcessHandle&gt;</code> for the parent process.
<code>long pid()</code>	Returns the native process ID (PID) of the process, which is assigned by the operating system. Note that a PID may be reused by operating systems if a process terminates, so two process handles having the same PID may not represent the same process.
<code>boolean supportsNormalTermination()</code>	Returns <code>true</code> if the implementation of <code>destroy()</code> normally terminates the process.

Table 11-3 lists the methods and descriptions of the `ProcessHandle.Info` nested interface. An instance of this interface contains snapshot information about a process. You can obtain a `ProcessHandle.Info` using the `info()` method of the `ProcessHandle` interface or the `Process` class. All methods in the interface return an `Optional`.



**Table 11-3.** *Methods in the ProcessHandle.Info Interface*

Method	Description
<code>Optional&lt;String[]&gt; arguments()</code>	Returns arguments of the process. The process may change the original arguments passed to it after startup. This method returns the changed arguments in that case.
<code>Optional&lt;String&gt; command()</code>	Returns the executable pathname of the process.
<code>Optional&lt;String&gt; commandLine()</code>	It is a convenience method for combining the command and arguments of a process. It returns the command line of the process by combining the values returned from the <code>command()</code> and <code>arguments()</code> methods if both methods return non-empty optionals.
<code>Optional&lt;Instant&gt; startInstant()</code>	Returns the start time of the process. If the operating system does not return a start time, it returns an empty <code>Optional</code> .
<code>Optional&lt;Duration&gt; totalCpuDuration()</code>	Returns the total CPU time used by the process. Note that a process may run for a long time and may use very little CPU time.
<code>Optional&lt;String&gt; user()</code>	Returns the user of the process.

It is time to see the `ProcessHandle` and `ProcessHandle.Info` interfaces in action. Listing 11-3 contains the code for a class named `CurrentProcessInfo`. Its `printInfo()` method takes a `ProcessHandle` as an argument and prints the details of the process. We also use this method in other examples to print the details of a process. The `main()` method gets the handle of the current process running the process, which is a Java process, and prints its details. You may get a different output. The output was generated when the program ran on Linux.

**Listing 11-3.** A `CurrentProcessInfo` Class That Prints the Details of the Current Process

```
// CurrentProcessInfo.java
package com.jdojo.process;
import java.time.Duration;
import java.time.Instant;
import java.time.ZoneId;
```

```

import java.time.ZonedDateTime;
import java.util.Arrays;
public class CurrentProcessInfo {
    public static void main(String[] args) {
        // Get the handle of the current process
        ProcessHandle current = ProcessHandle.current();
        // Print the process details
        printInfo(current);
    }
    public static void printInfo(ProcessHandle handle) {
        // Get the process ID
        long pid = handle.pid();
        // Is the process still running
        boolean isAlive = handle.isAlive();
        // Get other process info
        ProcessHandle.Info info = handle.info();
        String command = info.command().orElse("");
        String[] args = info.arguments()
            .orElse(new String[]{});
        String commandLine = info.commandLine()
            .orElse("");
        ZonedDateTime startTime = info.startInstant()
            .orElse(Instant.now())
            .atZone(ZoneId.systemDefault());
        Duration duration = info.totalCpuDuration()
            .orElse(Duration.ZERO);
        String owner = info.user().orElse("Unknown");
        long childrenCount = handle.children().count();
        // Print the process details
        System.out.printf("PID: %d\n", pid);
        System.out.printf("IsAlive: %b\n", isAlive);
        System.out.printf("Command: %s\n", command);
        System.out.printf("Arguments: %s\n",
            Arrays.toString(args));
        System.out.printf("CommandLine: %s\n",

```

```

        commandLine);
    System.out.printf("Start Time: %s%n", startTime);
    System.out.printf("CPU Time: %s%n", duration);
    System.out.printf("Owner: %s%n", owner);
    System.out.printf("Children Count: %d%n",
        childrenCount);
    }
}

```

PID: 4143

IsAlive: true

Command: /opt/jdk17/bin/java

Arguments: [-Dfile.encoding=UTF-8,  
 -classpath,  
 [<path-to-project>]/bin,  
 -XX:+ShowCodeDetailsInExceptionMessages,  
 com.jdojo.process.CurrentProcessInfo]

CommandLine: /opt/openjdk-16.36/bin/java

-Dfile.encoding=UTF-8  
 -classpath [<path-to-project>]/bin  
 -XX:+ShowCodeDetailsInExceptionMessages  
 com.jdojo.process.CurrentProcessInfo

Start Time: 2021-07-16T14:50:18.870+02:00  
 [Europe/Berlin]

CPU Time: PT0.06S

Owner: peter

Children Count: 0

## Comparing Processes

It is tricky to compare two processes for equality or order. You cannot rely on PIDs for equality of processes. Operating systems reuse PIDs after processes terminate. You may check the start time of processes along with the PIDs; if they are the same, the two processes may be the same. The `equals()` method of the default implementation of the

ProcessHandle interface checks for the following three pieces of information for two processes to be equal:

- The implementation class of the ProcessHandle interface must be the same for both processes.
- Processes must have the same PIDs.
- Processes must have been started at the same time.

---

**Note** Using the default implementation of the `compareTo()` method in the ProcessHandle interface is not very useful for ordering. It compares the PIDs of two processes.

---

## Creating a Process

You need to use an instance of the `ProcessBuilder` class to start a new native process. A `ProcessBuilder` manages a collection of native process attributes. Once you set all the attributes for the process, you can call its `start()` method to start a new native process. The attributes stored in the `ProcessBuilder` will be used to start the new process. You can call the `start()` method multiple times to start new processes using the attributes stored in the `ProcessBuilder`. The `start()` method returns an instance of the `Process` class that represents the new native process. You can use one of the following constructors to create an instance of the `ProcessBuilder` class:

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

The constructors let you specify the operating system program and arguments. Suppose you want to run the `java` program from inside `/opt/jdk17/bin` on Linux as follows:

```
/opt/jdk17/bin/java --version
```

You would create a `ProcessBuilder` to represent this command as follows:

```
ProcessBuilder pb = new ProcessBuilder(
    "/opt/jdk17/bin/java", "--version");
```

Using methods of the `ProcessBuilder` class, you can manage the following attributes of a process:

- A command
- An environment
- A working directory
- Standard I/O (`stdin`, `stdout`, and `stderr`)
- Redirection property for the standard error stream

A command is simply a list of strings representing the external program and its arguments. You can set the command in the constructor of the `ProcessBuilder` class. The following methods let you retrieve the command strings and set more command strings:

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

The `command()` method without any arguments returns the command strings already set in the `ProcessBuilder`. The `command()` method with a varargs argument lets you add more command strings. The following snippet of code creates a `ProcessBuilder` to launch JVM on Linux. It uses the `command()` method to set the command attribute:

```
ProcessBuilder pb = new ProcessBuilder()
    .command("/opt/jdk17/bin/java",
        "--module-path",
        "myModulePath",
        "--module",
        "myModule/className");
```

An environment is a list of system-dependent key-value pairs. It is initialized to a copy of the `Map<String, String>` returned from the `System.getenv()` static method. You need to use the `environment()` method of the `ProcessBuilder` class to get the `Map<String, String>` and add key-value pairs to the map. The following snippet of code shows you how to set the environment attributes for a `ProcessBuilder`:

```
ProcessBuilder pb = new ProcessBuilder("mycommand");
Map<String, String> env = pb.environment();
env.put("arg1", "value1");
env.put("arg2", "value2");
```

By default, the working directory for the new process would be the working directory of the current Java process, which is usually the directory named by the system property `user.dir`. The following methods in the `ProcessBuilder` class let you get and set the working directory:

- `File directory()`
- `ProcessBuilder directory(File directory)`

The following snippet of code shows you how to set the working directory to `/home/USER/mydir` on Linux:

```
ProcessBuilder pb = new ProcessBuilder("myCommand")
    .directory(new File("/home/USER/mydir"));
```

The new process created by the `start()` method of a `ProcessBuilder` is created as a child process of the current process, which is the Java process running the code. In other words, the current Java process is the parent process of the newly created process. The new process does not own a terminal or console for standard I/O (`stdin`, `stdout`, and `stderr`). By default, the I/O of the new process is connected to the parent process over a pipe. You have an option to set the standard I/O of the new process to the same as its parent process by calling the `inheritIO()` method of a `ProcessBuilder`. There are several `redirectXxx()` methods in the `ProcessBuilder` class to customize the standard I/O for the new process, for example, setting the standard error stream to a file, so all errors are logged to a file.

Once you have configured all attributes of the process, you can call `start()` to start the process:

```
// Start a new process
Process newProcess = pb.start();
```

You can call the `start()` method of the `ProcessBuilder` class multiple times to start multiple processes with the same attributes previously stored in it. This has a performance benefit that you can create one `ProcessBuilder` instance and reuse it to launch the same process multiple times.

You can obtain the process handle of a process using the `toHandle()` method of the `Process` class:

```
// Get the process handle
ProcessHandle handle = newProcess.toHandle();
```

You can use the process handle to destroy the process, wait for the process to finish, or query the process for its state and attributes such as its children, descendants, parents, CPU time used, etc. The information you get about a process and the control you have on a process depend on the operating system access controls.

It is tricky to come up with examples to create processes that will run on all operating systems. If you can run other examples in this book, it means that you have JDK17 installed on your machine. You can use the `java` program on your machine to launch other processes in the examples. You can use the `command` attribute of the current process, which is the current running `java` program, to get the path of the Java program on your machine, so the examples will work on all platforms.

Let's look at a few examples of creating native processes using the Java program. You can print the Java product version information to the standard output and standard error using the `-version` and `-version` options, respectively, as follows:

```
/opt/jdk17/bin/java --version
openjdk 17 2021-05-16
OpenJDK Runtime Environment (build 17+1-123)
OpenJDK 64-Bit Server VM (build 17+1-123, mixed mode, sharing)

/opt/jdk17/bin/java -version
openjdk 17 2021-05-16
OpenJDK Runtime Environment (build 17+1-123)
OpenJDK 64-Bit Server VM (build 17+1-123, mixed mode, sharing)
```

In the previous outputs, you do not see any difference as to where the output was printed. Both outputs are printed to the same console because, by default, both standard output and standard error are mapped to the console. However, you will see the difference when you try capturing the outputs from these two commands in a program.

Listing 11-4 shows a program that runs the `java -version` command to print the Java product information to the standard output.

#### **Listing 11-4.** Capturing the Output of a Native Process

```
// PipedIO.java
package com.jdojo.process;
import java.io.IOException;
public class PipedIO {
    public static void main(String[] args) {
```

```

// Get the path of the java program that started
// this program
String javaPath = ProcessHandle.current()
    .info()
    .command().orElse(null);
if(javaPath == null) {
    System.out.println(
        "Could not get the java command's path.");
    return;
}
// Configure the ProcessBuilder
ProcessBuilder pb =
    new ProcessBuilder(javaPath, "--version");
try {
    // Start a new java process
    Process p = pb.start();
} catch (IOException e) {
    e.printStackTrace();
}
}
}

```

When you run the program `ProcessIO` class, it does not print anything. Where did the output go? The program created a new process, and the standard output of the process was connected to the parent process over a pipe. If you want to access the output, you need to read from the appropriate pipe. When the standard I/O of the new process is piped to the parent process, you can use the following methods of the `Process` to get the I/O streams of the new process:

- `OutputStream getOutputStream()`
- `InputStream getInputStream()`
- `InputStream getErrorStream()`

The `OutputStream` returned from the `getOutputStream()` method is connected to the standard input stream of the new process. Writing to this output stream will be piped to the standard input of the new process.



The `InputStream` returned from the `getInputStream()` is connected to the standard output of the new process. If you want to capture the standard output of the new process, you need to read from this input stream.

The `InputStream` returned from the `getErrorStream()` is connected to the standard error of the new process. If you want to capture the standard error of the new process, you need to read from this input stream. Sometimes, you want to merge the output to the standard output and standard error into one destination. It gives the exact sequence of output and the error for easier troubleshooting issues. You can call the `redirectErrorStream(true)` method of the `ProcessBuilder` to send the data written to the standard error to the standard output. I show examples of this kind shortly.

---

**Note** You have options to redirect the standard I/O of a new process to other destinations such as a file, and in that case, the `getOutputStream()`, `getInputStream()`, and `getErrorStream()` methods return `null`.

---

The program in Listing 11-5 fixes the problem of not getting any output in the `PipedIO` class. It reads and prints the data written to the standard output stream in the pipe.

**Listing 11-5.** Capturing the Output of a Native Process

```
// CapturePipedIO.java
package com.jdojo.process;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class CapturePipedIO {
    public static void main(String[] args) {
        // Get the path of the java program that started
        // this program
        String javaPath = ProcessHandle.current()
            .info()
            .command().orElse(null);
        if (javaPath == null) {
            System.out.println(
```

```

        "Could not get the java command's path.");
    return;
}
// Configure the ProcessBuilder
ProcessBuilder pb =
    new ProcessBuilder(javaPath, "--version");
try {
    // Start a new java process
    Process p = pb.start();
    // Read and print the standard output stream
    // of the process
    try (BufferedReader input =
        new BufferedReader(
            new InputStreamReader(
                p.getInputStream()))) {
        String line;
        while ((line = input.readLine()) != null) {
            System.out.println(line);
        }
    }
} catch (IOException e) {
    e.printStackTrace();
}
}
}

```

```
openjdk 17 2021-05-16
```

```
OpenJDK Runtime Environment (build 17+1-123)
```

```
OpenJDK 64-Bit Server VM (build 17+1-123, mixed mode, sharing)
```

If you run the java command with a `-version` option, the output is written to the standard error. If you change the option from `-version` to `-version` in Listing 11-5, you will not get any output again because the output will be piped to the standard error stream. You have two ways to fix this:

- In the program, read from the `InputStream` returned from the `getErrorStream()` method of the `Process` instead of the `InputStream` from the `getInputStream()` method.
- Redirect the error stream to the standard output stream and keep reading from the standard output.

The following snippet of code creates a `ProcessBuilder` with the `java -version` command and redirects the error stream in the standard output:

```
// Configure the ProcessBuilder
ProcessBuilder pb =
    new ProcessBuilder(javaPath, "-version")
    .redirectErrorStream(true);
```

If you change the statement that creates the `ProcessBuilder` in Listing 11-5 to this statement, your program will work fine.

A new process can also inherit the standard I/O of the parent process. If you want to set all I/O destinations of the new process to the same as the current process, use the `inheritIO()` method of the `ProcessBuilder`, as shown:

```
// Configure the ProcessBuilder inheriting parent's I/O
ProcessBuilder pb =
    new ProcessBuilder(javaPath, "--version")
    .inheritIO();
```

If you change the code in Listing 11-4 to match the previous snippet of code, you will see the output.

The `ProcessBuilder.Redirect` nested class represents the source of the input and destination of the outputs of the new process created by the `ProcessBuilder`. The class defined the following three constants of the `ProcessBuilder.Redirect` type:

- `ProcessBuilder.Redirect.DISCARD`: Discards the outputs of the new process
- `ProcessBuilder.Redirect.INHERIT`: Indicates that the input source or output destination of the new process will be the same as that of the current process
- `ProcessBuilder.Redirect.PIPE`: Indicates that the new process will be connected to the current process over a pipe, which is the default

You can also redirect the input and outputs of the new process to a file using the following methods of the `Process.Redirect` class:

- `ProcessBuilder.Redirect appendTo(File file)`
- `ProcessBuilder.Redirect from(File file)`
- `ProcessBuilder.Redirect to(File file)`

In the previous snippet of code, you saw how to use the `inheritIO()` method of the `ProcessBuilder` class to let the new process have the same standard I/O as the current process. You can rewrite that code as follows:

```
// Configure the ProcessBuilder inheriting parent's I/O
ProcessBuilder pb =
    new ProcessBuilder(javaPath, "--version")
        .redirectInput(ProcessBuilder.Redirect.INHERIT)
        .redirectOutput(ProcessBuilder.Redirect.INHERIT)
        .redirectError(ProcessBuilder.Redirect.INHERIT);
```

The following snippet of code redirects the standard output of the new process to a file named `java_product_details.txt` in the current directory:

```
// Configure the ProcessBuilder
ProcessBuilder pb =
    new ProcessBuilder(javaPath, "--version")
        .redirectOutput(
            ProcessBuilder.Redirect.to(
                new File("java_product_details.txt")));
```

Let's look at a little complex example that will explore more information about new native processes. Listing 11-6 contains the code for a class named `Job`. Its `main()` method expects two arguments: sleep interval and sleep duration in seconds. If they are not passed, the method uses 5 seconds and 60 seconds as the default values. In the first part, the method attempts to extract first and second arguments, if specified. In the second part, it gets the process handle of the current process executing this method using the `ProcessHandle.current()` method. It reads the PID of the current process and prints a message including the PID, sleep interval, and sleep duration. In the end, it starts a for loop and keeps sleeping for the sleep interval until the sleep duration is reached. In every iteration of the loop, it prints a message.

**Listing 11-6.** The Declaration of a Class Named Job

```
// Job.java
package com.jdojo.process;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.TimeUnit;
import java.util.stream.Collectors;
/**
 * An instance of this class is used as a job that sleeps
 * at a regular interval up to a maximum duration. The
 * sleep interval in seconds can be specified as the first
 * argument and the sleep duration as the second argument
 * while running this class. The default sleep interval
 * and sleep duration are 5 seconds and 60 seconds,
 * respectively. If these values are less than zero, zero
 * is used instead.
 */
public class Job {
    // The job sleep interval
    public static final long DEFAULT_SLEEP_INTERVAL = 5;
    // The job sleep duration
    public static final long DEFAULT_SLEEP_DURATION = 60;
    public static void main(String[] args) {
        long sleepInterval = DEFAULT_SLEEP_INTERVAL;
        long sleepDuration = DEFAULT_SLEEP_DURATION;
        // Get the passed in sleep interval
        if (args.length >= 1) {
            sleepInterval = parseArg(args[0],
                DEFAULT_SLEEP_INTERVAL);
            if (sleepInterval < 0) {
                sleepInterval = 0;
            }
        }
    }
}
```

```

// Get the passed in the sleep duration
if (args.length >= 2) {
    sleepDuration = parseArg(args[1],
        DEFAULT_SLEEP_DURATION);
    if (sleepDuration < 0) {
        sleepDuration = 0;
    }
}
long pid = ProcessHandle.current().pid();
System.out.printf(
    "Job (pid=%d) info: Sleep Interval"
    + "=%d seconds, Sleep Duration=%d "
    + "seconds.%n",
    pid, sleepInterval, sleepDuration);
for (long sleptFor = 0; sleptFor < sleepDuration;
    sleptFor += sleepInterval) {
    try {
        System.out.printf(
            "Job (pid=%d) is going to"
            + " sleep for %d seconds.%n",
            pid, sleepInterval);
        // Sleep for the sleep interval
        TimeUnit.SECONDS.sleep(sleepInterval);
    } catch (InterruptedException ex) {
        System.out.printf("Job (pid=%d) was "
            + "interrupted.%n", pid);
    }
}
}
}
/**
 * Starts a new JVM to run the Job class.
 *
 * @param sleepInterval The sleep interval when the
 * Job class is run. It is passed to the JVM as the
 * first argument.

```

```

* @param sleepDuration The sleep duration for the
*   Job class. It is passed to the JVM as the
*   second argument.
* @return The new process reference of the newly
*   launched JVM or null if the JVM
*   cannot be launched.
*/
public static Process startProcess(long sleepInterval,
    long sleepDuration) {
    // Store the command to launch a new JVM in a
    // List<String>
    List<String> cmd = new ArrayList<>();
    // Add command components in order
    addJvmPath(cmd);
    addModulePath(cmd);
    addClassPath(cmd);
    addMainClass(cmd);
    // Add arguments to run the class
    cmd.add(String.valueOf(sleepInterval));
    cmd.add(String.valueOf(sleepDuration));
    // Build the process attributes
    ProcessBuilder pb = new ProcessBuilder()
        .command(cmd)
        .inheritIO();
    String commandLine = pb.command()
        .stream()
        .collect(Collectors.joining(" "));
    System.out.println(
        "Command used:\n" + commandLine);
    // Start the process
    Process p = null;
    try {
        p = pb.start();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

```

        return p;
    }
/**
 * Used to parse the arguments passed to the JVM,
 * which in turn is passed to the main() method.
 *
 * @param valueStr The string value of the argument
 * @param defaultValue The default value of the
 *     argument if the valueStr is not an integer.
 * @return valueStr as a long or the defaultValue if
 *     valueStr is not an integer.
 */
private static long parseArg(String valueStr,
    long defaultValue) {
    long value = defaultValue;
    if (valueStr != null) {
        try {
            value = Long.parseLong(valueStr);
        } catch (NumberFormatException e) {
            // no action needed
        }
    }
    return value;
}
/**
 * Adds the JVM path to the command list. It first
 * attempts to use the command attribute of the
 * current process; failing that it relies on the
 * java.home system property.
 *
 * @param cmd The command list
 */
private static void addJvmPath(List<String> cmd) {
    // First try getting the command to run the
    // current JVM

```



```

    String jvmPath = ProcessHandle.current()
        .info()
        .command().orElse("");
    if (jvmPath.length() > 0) {
        cmd.add(jvmPath);
    } else {
        // Try composing the JVM path using the
        // java.home system property
        final String FILE_SEPARATOR =
            System.getProperty("file.separator");
        jvmPath = System.getProperty("java.home")
            + FILE_SEPARATOR + "bin"
            + FILE_SEPARATOR + "java";
        cmd.add(jvmPath);
    }
}
/**
 * Adds a module path to the command list.
 *
 * @param cmd The command list
 */
private static void addModulePath(List<String> cmd) {
    String modulePath
        = System.getProperty("jdk.module.path");
    if (modulePath != null
        && modulePath.trim().length() > 0) {
        cmd.add("--module-path");
        cmd.add(modulePath);
    }
}
/**
 * Adds class path to the command list.
 *
 * @param cmd The command list
 */

```

```

private static void addClassPath(List<String> cmd) {
    String classPath =
        System.getProperty("java.class.path");
    if (classPath != null
        && classPath.trim().length() > 0) {
        cmd.add("--class-path");
        cmd.add(classPath);
    }
}
/**
 * Adds a main class to the command list. Adds
 * module/className or just className depending on
 * whether the Job class was loaded in a named
 * module or unnamed module
 *
 * @param cmd The command list
 */
private static void addMainClass(List<String> cmd) {
    Class<Job> cls = Job.class;
    String className = cls.getName();
    Module module = cls.getModule();
    if (module.isNamed()) {
        String moduleName = module.getName();
        cmd.add("--module");
        cmd.add(moduleName + "/" + className);
    } else {
        cmd.add(className);
    }
}
}

```

The `Job` class contains a `startProcess(long sleepInterval, long sleepDuration)` method that starts a new process. It launches a JVM with the `Job` class as the main class. It passes the sleep interval and duration to the JVM as arguments. The method attempts to build a command to launch the `java` command from

the `JDK_HOME\bin` directory. If the `Job` class were loaded in a named module, it would build a command like this:

```
JDK_HOME/bin/java --module-path <module-path> \
--module jdojo.process/com.jdojo.process.Job \
<sleepInterval> <sleepDuration>
```

If the `Job` class were loaded in an unnamed module, it would attempt to build a command like this:

```
JDK_HOME/bin/java \
-class-path <class-path> \
com.jdojo.process.Job \
<sleepInterval> <sleepDuration>
```

The `startProcess()` method prints the command used to start a process, attempts to start the process, and returns the process reference.

The `addJvmPath()` method adds the JVM path to the command list. It attempts to get the command for the current JVM process to use as the JVM path for the new process. If it is not available, it attempts to build it from the `java.home` system property.

The `Job` class contains several utility methods that are used to compose parts of commands and parse the arguments passed to the `main()` method. Refer to their Javadoc for descriptions.

If you want to start a new process that should run for 15 seconds and wake up every 5 seconds, you can do so using the `startProcess()` method of the `Job` class:

```
// Start a process that runs for 15 seconds
Process p = Job.startProcess(5, 15);
```

You can print the process details using the `printInfo()` method of the `CurrentProcessInfo` class that you created in Listing 11-3:

```
// Get the handle of the current process
ProcessHandle handle = p.toHandle();
// Print the process details
CurrentProcessInfo.printInfo(handle);
```

You can use the returned value of the `onExit()` method of the `ProcessHandle` to run a task when the process terminates:

```
CompletableFuture<ProcessHandle> future = handle.onExit();
// Print a message when process terminates
future.thenAccept((ProcessHandle ph) -> {
    System.out.printf(
        "Job (pid=%d) terminated.%n", ph.pid());
});
```

You can wait for the new process to terminate like so:

```
// Wait for the process to terminate
future.get();
```

In this example, `future.get()` will return the `ProcessHandle` of the process. I did not use the return value, because I already had it in the `handle` variable.

Listing 11-7 contains the code for a `StartProcessTest` class that shows you how to create a new process using the `Job` class. In its `main()` method, it creates a new process, prints process details, adds a shutdown task to the process, waits for the process to terminate, and prints the process details again. Note that the process runs for 15 seconds, but it uses only 0.359375 seconds of CPU time because most of the time the main thread of the process was sleeping. You may get a different output. The output was generated when the program ran on Linux.

**Listing 11-7.** A `StartProcessTest` Class That Creates New Processes

```
// StartProcessTest.java
package com.jdojo.process;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
public class StartProcessTest {
    public static void main(String[] args) {
        // Start a process that runs for 15 seconds
        Process p = Job.startProcess(5, 15);
        if (p == null) {
            System.out.println(
                "Could not create a new process.");
        }
    }
}
```

```

        return;
    }
    // Get the handle of the current process
    ProcessHandle handle = p.toHandle();
    // Print the process details
    CurrentProcessInfo.printInfo(handle);
    CompletableFuture<ProcessHandle> future =
        handle.onExit();
    // Print a message when process terminates
    future.thenAccept((ProcessHandle ph) -> {
        System.out.printf(
            "Job (pid=%d) terminated.%n",
            ph.pid());
    });
    try {
        // Wait for the process to complete
        future.get();
    } catch (InterruptedException
            | ExecutionException e) {
        e.printStackTrace();
    }
    // Print process details again
    CurrentProcessInfo.printInfo(handle);
}
}

```

Command used:

```

/opt/jdk17/bin/java
  --class-path /[<path-to-project>]/bin
  com.jdojo.process.Job 5 15

```

PID: 8701

IsAlive: true

Command: /opt/jdk17/bin/java

Arguments: [  
 --class-path,

```
    / [<path-to-project> ] / bin,  
    com.jdojo.process.Job,  
    5, 15 ]  
CommandLine: /opt/jdk17/bin/java  
    --class-path / [<path-to-project> ] / bin  
    com.jdojo.process.Job  
    5 15  
Start Time: 2021-07-16T18:11:42.510+02:00  
    [Europe/Berlin]  
CPU Time: PT0.01S  
Owner: peter  
Children Count: 0  
Job (pid=8701) info:  
    Sleep Interval=5 seconds, Sleep Duration=15 seconds.  
Job (pid=8701) is going to sleep for 5 seconds.  
Job (pid=8701) is going to sleep for 5 seconds.  
Job (pid=8701) is going to sleep for 5 seconds.  
Job (pid=8701) terminated.  
PID: 8701  
IsAlive: false  
Command:  
Arguments: []  
CommandLine:  
Start Time: 2021-07-16T18:11:58.489975569+02:00  
    [Europe/Berlin]  
CPU Time: PT0S  
Owner: Unknown  
Children Count: 0
```

## Obtaining a Process Handle

There are several ways to get the handle of a native process. For a process created by the Java code, you can get a `ProcessHandle` using the `toHandle()` method of the `Process` class. Native processes can also be created from outside the JVM. The `ProcessHandle` interface contains the following methods to get the handle of a native process:

- `static Optional<ProcessHandle> of(long pid)`
- `static ProcessHandle current()`
- `Optional<ProcessHandle> parent()`
- `Stream<ProcessHandle> children()`
- `Stream<ProcessHandle> descendants()`
- `static Stream<ProcessHandle> allProcesses()`

The `of()` static method returns an `Optional<ProcessHandle>` for the specified `pid`. If there is no process with this `pid`, an empty `Optional` is returned. To use this method, you need to know the PID of the process:

```
// Get the process handle of the process with the pid
// of 1234
Optional<ProcessHandle> handle = ProcessHandle.of(1234L);
```

The `current()` static method returns the handle of the current process, which is always the Java process executing the code. You have already seen an example of this in [Listing 11-3](#).

The `parent()` method returns the handle of the parent process. It returns an empty `Optional` if the process does not have a parent or the parent cannot be retrieved.

The `children()` method returns a snapshot of all direct child processes of the process. There is no guarantee that a process returned by this method is still alive. Note that a process that's not alive does not have children.

The `descendants()` method returns a snapshot of all child processes of the process, direct or indirect.

The `allProcesses()` method returns a snapshot of all processes that are visible to this process. There is no guarantee that the stream contains all process in the operating system at the time the stream is processed. Processes may have been terminated or

created after the snapshot was taken. The following snippet of code prints the PIDs of all processes sorted by their PIDs:

```
System.out.printf("All processes PIDs:%n");
ProcessHandle.allProcesses()
    .map(ph -> ph.pid())
    .sorted()
    .forEach(System.out::println);
```

You can compute different types of statistics for all running processes. You can also create a task manager in Java that displays a UI showing all running processes and their attributes. Listing 11-8 shows how to get the longest running process details and the process that used the CPU time the most. I compared the start time of the processes to get the longest running process and the total CPU duration to get the process that used the CPU time the most. You may get a different output. I got this output when I ran the program on Linux.

**Listing 11-8.** Computing Process Statistics

```
// ProcessStats.java
package com.jdojo.process;
import java.time.Duration;
import java.time.Instant;
public class ProcessStats {
    public static void main(String[] args) {
        System.out.printf("Longest CPU User Process:%n");
        ProcessHandle.allProcesses()
            .max(ProcessStats::compareCpuTime)
            .ifPresent(CurrentProcessInfo::printInfo);
        System.out.printf("%nLongest Running Process:%n");
        ProcessHandle.allProcesses()
            .max(ProcessStats::compareStartTime)
            .ifPresent(CurrentProcessInfo::printInfo);
    }
    public static int compareCpuTime(ProcessHandle ph1,
        ProcessHandle ph2) {
        return ph1.info()
```



```

        .totalCpuDuration()
        .orElse(Duration.ZERO)
        .compareTo(ph2.info()
            .totalCpuDuration()
            .orElse(Duration.ZERO));
    }
    public static int
    compareStartTime(ProcessHandle ph1,
        ProcessHandle ph2) {
        return ph1.info()
            .startInstant()
            .orElse(Instant.now())
            .compareTo(ph2.info()
                .startInstant()
                .orElse(Instant.now()));
    }
}

```

Longest CPU User Process:

PID: 2323

IsAlive: true

Command: /usr/lib/tracker/tracker-miner-fs

Arguments: []

CommandLine: /usr/lib/tracker/tracker-miner-fs

Start Time: 2021-07-16T13:43:03.590+02:00[Europe/Berlin]

CPU Time: PT14M35.72S

Owner: peter

Children Count: 0

Longest Running Process:

PID: 9019

IsAlive: true

Command: /opt/openjdk-16.36/bin/java

Arguments: [

-Dfile.encoding=UTF-8,

-classpath,

```
[...],
-XX:+ShowCodeDetailsInExceptionMessages,
com.jdojo.process.ProcessStats]
CommandLine: /opt/jdk17/bin/java
-Dfile.encoding=UTF-8
-classpath [...]
-XX:+ShowCodeDetailsInExceptionMessages
com.jdojo.process.ProcessStats
Start Time: 2021-07-16T19:02:01.020+02:00[Europe/Berlin]
CPU Time: PT0.3S
Owner: peter
Children Count: 0
```

## Terminating Processes

You can terminate a process using the `destroy()` or `destroyForcibly()` method of the `ProcessHandle` interface and the `Process` class. Both methods return `true` if the request to terminate the process was successful, `false` otherwise. The `destroy()` method requests a normal termination, whereas the `destroyForcibly()` method requests a forced termination. It is possible for the `isAlive()` method to return `true` for a brief period after a request to terminate the process has been made.

---

**Note** You cannot terminate the current process. Calling the `destroy()` or the `destroyForcibly()` method on the current process throws an `IllegalStateException`. The operating system access controls may prevent a process from being terminated.

---

A normal termination of a process lets the process terminate cleanly. A forced termination of a process terminates the process immediately. Whether a process is normally terminated is implementation dependent. You can use the `supportsNormalTermination()` method of the `ProcessHandle` interface and the `Process` class to check if a process supports normal termination. The method returns `true` if the process supports normal termination, `false` otherwise.

Calling one of these methods to terminate a process that has already been terminated results in no action. The `CompletableFuture<Process>` returned from `onExit()` of the `Process` class and the `CompletableFuture<ProcessHandle>` returned from `onExit()` of the `ProcessHandle` interface are completed when the process terminates.

## Managing Process Permissions

When you ran the examples in the previous sections, I assumed that there was no Java security manager installed. If a security manager is installed, appropriate permissions need to be granted to start, manage, and query native processes:

- If you are creating a new process, you need to have `FilePermission(cmd,"execute")` permission, where `cmd` is the absolute path of the command that will create the process. If `cmd` is not an absolute path, you need to have `FilePermission("<<ALL FILES>>","execute")` permission.
- To query the state of native processes and destroy the process using the methods in the `ProcessHandle` interface, the application needs to have `RuntimePermission("manageProcess")` permission.

Listing 11-9 contains a program that gets a process count and creates a new process. It repeats these two tasks without a security manager and with a security manager.

### **Listing 11-9.** Managing Processes with a Security Manager

```
// ManageProcessPermission.java
package com.jdojo.process;
import java.util.concurrent.ExecutionException;
public class ManageProcessPermission {
    public static void main(String[] args) {
        // Get the process count
        long count = ProcessHandle.allProcesses().count();
        System.out.printf("Process Count: %d\n", count);
        // Start a new process
        Process p = Job.startProcess(1, 3);
        try {
            p.toHandle().onExit().get();
        }
```

```

    } catch (InterruptedException
        | ExecutionException e) {
        System.out.println(e.getMessage());
    }
    // Install a security manager
    SecurityManager sm = System.getSecurityManager();
    if (sm == null) {
        System.setSecurityManager(
            new SecurityManager());
        System.out.println(
            "A security manager is installed.");
    }
    // Get the process count
    try {
        count = ProcessHandle.allProcesses().count();
        System.out.printf("Process Count: %d%n",
            count);
    } catch (RuntimeException e) {
        System.out.println(
            "Could not get a process count: " +
            e.getMessage());
    }
    // Start a new process
    try {
        p = Job.startProcess(1, 3);
        p.toHandle().onExit().get();
    } catch (InterruptedException
        | ExecutionException
        | RuntimeException e) {
        System.out.println(
            "Could not start a new process: " +
            e.getMessage());
    }
}
}

```

Try running the `ManageProcessPermission` class using the following command assuming that you have not changed any Java policy files:

```
/opt/jdk17/bin/java \
-Dfile.encoding=UTF-8 \
-classpath /[<path-to-project>]/bin \
-XX:+ShowCodeDetailsInExceptionMessages \
com.jdojo.process.ManageProcessPermission

Process Count: 332
Command used:
/opt/jd17/bin/java
  --class-path [...] com.jdojo.process.Job 1 3
Job (pid=3858) info: Sleep Interval=1 seconds,
  Sleep Duration=3 seconds.
Job (pid=3858) is going to sleep for 1 seconds.
Job (pid=3858) is going to sleep for 1 seconds.
Job (pid=3858) is going to sleep for 1 seconds.
A security manager is installed.
Could not get a process count: access denied
  ("java.lang.RuntimePermission" "manageProcess")
Could not start a new process: access denied
  ("java.lang.RuntimePermission" "manageProcess")
```

You may get a different output. The output indicates that you were able to get the process count and create a new process before a security manager was installed. After the security manager was installed, the Java runtime threw exceptions while requesting the process count and creating a new process. To fix the problem, you need to grant the following permissions:

- The `"manageProcess"` `RuntimePermission`, which will allow the application to query the native process and create a new process
- The `"execute"` `FilePermission` on the Java command path, which will allow launching the JVM
- The `"read"` `PropertyPermission` on the `"jdk.module.path"` and `"java.class.path"` system properties, so the `Job` class can read these properties while building the command line to launch the JVM

Listing 11-10 contains a script to grant these four permissions to all code. You need to add this script to the `JDK_HOME/conf/security/java.policy` file on your machine. The path to the Java launcher is `/opt/jdk17/bin/java`, and it is valid on Linux only if you have installed JDK17 in the `/opt/jdk17` directory. For all other platforms and JDK installations, modify this path to point to the correct Java launcher on your machine.

**Listing 11-10.** Addendum to the `JDK_HOME/conf/security/java.policy` File

```
grant {
    permission java.lang.RuntimePermission
        "manageProcess";
    permission java.io.FilePermission
        "/opt/jdk17/bin/java", "execute";
    permission java.util.PropertyPermission
        "jdk.module.path", "read";
    permission java.util.PropertyPermission
        "java.class.path", "read";
};
```

If you run the `ManageProcessPermission` class again using the same command, you should get output similar to the following:

```
/opt/jdk17/bin/java \
-Dfile.encoding=UTF-8 \
-classpath /[<path-to-project>]/bin \
-XX:+ShowCodeDetailsInExceptionMessages \
com.jdojo.process.ManageProcessPermission

Process Count: 330
Command used:
/opt/jdk17/bin/java
--class-path [...]
com.jdojo.process.Job 1 3
Job (pid=6093) info: Sleep Interval=1 seconds,
Sleep Duration=3 seconds.
Job (pid=6093) is going to sleep for 1 seconds.
```

Job (pid=6093) is going to sleep for 1 seconds.

Job (pid=6093) is going to sleep for 1 seconds.

A security manager is installed.

Process Count: 330

Command used:

/opt/jdk17/bin/java

--class-path [...]

com.jdojo.process.Job 1 3

Job (pid=6114) info: Sleep Interval=1 seconds,

Sleep Duration=3 seconds.

Job (pid=6114) is going to sleep for 1 seconds.

Job (pid=6114) is going to sleep for 1 seconds.

Job (pid=6114) is going to sleep for 1 seconds.

## Summary

The Process API consists of classes and interfaces to work with native processes. Java SE has provided the Process API since version 1.0 through the `Runtime` and `Process` classes. It allowed you to create new native processes, manage their I/O streams, and destroy them. Later versions of Java SE improved the API, with an interface named `ProcessHandle` that represents a process handle. You can use the process handle to query and manage a native process.

The following classes and interfaces comprise the Process API: `Runtime`, `ProcessBuilder`, `ProcessBuilder.Redirect`, `Process`, `ProcessHandle`, and `ProcessHandle.Info`.

The `exec()` method of the `Runtime` class is used to start a native process. The `start()` method of the `ProcessBuilder` class is preferred over the `exec()` method of the `Runtime` class to start a process. An instance of the `ProcessBuilder.Redirect` class represents a source of input of a process or a destination output of a process.

By default, the standard I/O of the new process is connected to the current process over a pipe. You need to read and write the streams associated with the pipe to access the standard I/O of the new process. You have options to set the standard I/O of the new process to the same as that of the current process or redirect the I/O to other sources/destinations such as a file.

An instance of the `Process` class represents a native process created by a Java program.

An instance of the `ProcessHandle` interface represents a process created by a Java program or by other means; it was added in Java 9 and provides several methods to query and manage processes. An instance of the `ProcessHandle.Info` interface represents snapshot information of a process; it can be obtained using the `info()` method of the `Process` class or `ProcessHandle` interface. If you have a `Process` instance, use its `toHandle()` method to get a `ProcessHandle`.

The `onExit()` method of the `ProcessHandle` interface returns a `CompletableFuture<ProcessHandle>` for the termination of the process. You can use the returned object to add a task that will be executed when the process terminates. Note that you cannot use this method on the current process.

If a security manager is installed, the application needs to have a "manageProcess" `RuntimePermission` to query and manage native processes and an "execute" `FilePermission` on the command file of the process that is started from the Java code.

## Exercises

### Exercise 1

What is the Process API?

### Exercise 2

What does an instance of the `Runtime` class represent?

### Exercise 3

How do you get an instance of the `Runtime` class?

### Exercise 4

How do you use the `ProcessBuilder` class? What method of this class is used to start a new native process?

### Exercise 5

What does an instance of the `Process` class represent?

### Exercise 6

What does an instance of the `ProcessHandle` interface represent? How do you obtain a `ProcessHandle` from a `Process`?

### Exercise 7

How do you get the handle of the current process representing the running Java program?



**Exercise 8**

What does an instance of the `ProcessHandle.Info` interface represent?

**Exercise 9**

What is the default standard I/O of the new process created by the `start()` method of the `ProcessBuilder` class?

**Exercise 10**

Can you terminate the current Java program using the Process API?