DU1 – The Java Runtime Class

Introduction

Every Java application has a single instance of **Runtime** that allows the application to interface with the environment in which the application is running.

Java Runtime class is used to interact with java runtime environment. Java Runtime class provides methods to execute a process, invoke GC, get total and free memory, etc. There is only one instance of **java.lang.Runtime** class available for one java application. The Runtime class is a **singleton class** that provides an interface to the underlying **operating system**.

The **Runtime.getRuntime()** method returns the singleton instance of Runtime class.

Specification java.lang.Runtime.

Important methods of Java Runtime class

The following **table** shows some of the commonly used methods of **Runtime** class.

Method	**Description**
public static Runtime getRuntime()	Returns the instance of Runtime class.
public void exit(int status)	Terminates the current virtual machine.
Public Process exec(String[] cmdarray) throws IOException	Executes the specified command and arguments in a separate process.
public long freeMemory()	Returns the amount of free memory in JVM.
public long totalMemory()	Returns the amount of total memory in JVM.
public long maxMemory()	Returns the maximum amount of memory that the JVM will attempt to use.
public void gc()	Runs the garbage collector.
public int availableProcessors()	Returns the number of processors available to the JVM.

Runtime.exec() method overview

In Java the Runtime.exec() method is used to execute external commands, applications, scripts, or processes from a Java application.

The Runtime.exec() method creates a native process representing the command or application and returns a Process object representing the newly created process.

This method returns a **Process object** for managing the subprocess.

You can retrieve the output stream, input stream, and error stream of the subprocess using the Process object.

The subprocess is not synchronized with the Java application, so you might want to use methods like **waitFor()** on the **Process** object to have your Java program wait for the subprocess to finish.

Be cautious while executing arbitrary commands, especially if they are derived from untrusted sources.

The following code creates a new **Process** that runs the **Notepad++** editor.

```java
package org.example;
import java.io.IOException;

public class NotepadPlusPlus {
    public static void main(String[] args) throws IOException {
        Runtime runtime=Runtime.getRuntime();
        String[] command={"C:/Program Files/Notepad++/Notepad++.exe"};
        Process process=runtime.exec(command);
    }
}
```

The following code opens a file **info.txt** created in the **resources folder** of the project with **Notepad++**. Maven resources folder is used to store all your project resources files like xml files, images, text files and etc. The **default Maven resources folder** is located at "**yourproject/src/main/resources**".

```java
package org.example;

import java.io.IOException;

public class NotepadPlusPlusFile {
    public static void main(String[] args) throws IOException {
        Runtime runtime=Runtime.getRuntime();
        String[] command={"C:/Program
Files/Notepad++/Notepad++.exe","./src/main/resources/info.txt"};
        Process process=runtime.exec(command);
    }
}
```

Getting standard output

If we execute a **process** that produce some **output** as a result, we need to capture the output by creating a **BufferedReader** that wraps the **InputStream** returned from the **Process**.

```
BufferedReader reader = new BufferedReader(new
InputStreamReader(process.getInputStream()));
```

Then we can invoke the **readLine()** method of the reader to read the output line by line, sequentially.

```
String line;
        while ((line = reader.readLine()) != null) {
            System.out.println(line);
        }
        reader.close();
```

We can also use a Scanner to read the command’s output.

```
Scanner scanner = new Scanner(process.getInputStream());
while (scanner.hasNextLine()) {
        System.out.println(scanner.nextLine());
}
scanner.close();
```

The following example shows how to execute the date command to show the date using the Scanner. To print today’s date on the command prompt, we can run date /t.

```
package org.example;

import java.io.IOException;
import java.util.Scanner;

public class CommandDate {
    public static void main(String[] args) throws IOException {
        String[] command = {"cmd", "/c", "date","/T"};
        Runtime runtime=Runtime.getRuntime();
        Process process=runtime.exec(command);
        Scanner scanner = new Scanner(process.getInputStream());
        while (scanner.hasNextLine()) {
            System.out.println(scanner.nextLine());
        }
        scanner.close();
    }
}
```

The following code executes **date/t** using **BufferedReader** to get the standard output.

```
package org.example;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class CommandDateBR {
    public static void main(String[] args) throws IOException {
        String[] command = {"cmd", "/c", "date","/T"};
        Runtime runtime=Runtime.getRuntime();
        Process process=runtime.exec(command);
        BufferedReader reader = new BufferedReader(new
InputStreamReader(process.getInputStream()));
        String line;
        while ((line = reader.readLine()) != null) {
            System.out.println(line);
        }
        reader.close();
    }
}
```

Getting error output

A command is not always executed successfully, because there would be a case in which the command encounters an error. And typically, error messages are sent to the error stream.

The following program reads a command to be executed from standard input and displays error messages to the standard error.

```
package org.example;

import java.io.IOException;
import java.util.Scanner;

public class CommandHelp {
    public static void main(String[] args) throws IOException,
InterruptedException {
        Scanner in=new Scanner(System.in);
        String data=in.nextLine();
        data="cmd /c "+data;
        String[] command=data.split("\\s+");
        Runtime runtime=Runtime.getRuntime();
        Process process=runtime.exec(command);
        Scanner scanner = new Scanner(process.getInputStream());
        while (scanner.hasNextLine()) {
            System.out.println(scanner.nextLine());
        }
        scanner.close();
        int exitCode=process.waitFor();
        if (exitCode!=0){
            Scanner errorScanner = new Scanner(process.getErrorStream());
            String line;
            while (errorScanner.hasNextLine()) {
                System.out.println(errorScanner.nextLine());
            }
            errorScanner.close();
        }
    }
}
```

Sending input

For interactive programs which need inputs, we can feed the inputs for the command by obtaining the **OutputStream** returned by the **Process**.

The following program attempts to change system date on Windows to 24-09-24 (in dd-mm-yy format).

```
package org.example;

import java.io.BufferedWriter;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.util.Scanner;

public class CommandDateInt {
    public static void main(String[] args) throws IOException,
InterruptedException {
        String[] command = {"cmd","/C","date"};
        Runtime runtime=Runtime.getRuntime();
        Process process=runtime.exec(command);
        BufferedWriter writer = new BufferedWriter(new
OutputStreamWriter(process.getOutputStream()));
        writer.write("24-09-24");
        writer.close();

        Scanner scanner = new Scanner(process.getInputStream());
        while (scanner.hasNextLine()) {
            System.out.println(scanner.nextLine());
        }
        scanner.close();
        int exitCode=process.waitFor();
        if (exitCode!=0){
            Scanner errorScanner = new Scanner(process.getErrorStream());
            String line;
            while (errorScanner.hasNextLine()) {
                System.out.println(errorScanner.nextLine());
            }
            errorScanner.close();
        }
    }
}
```

Waiting for the process to terminate

We can make the calling thread to wait until the process has terminated, by invoking the waitFor() method on the Process object.

```
int exitCode=process.waitFor();
if (exitCode!=0){
    System.out.println("Abnormal process termination");
}
```