HOL6427: Uncover JDK 8 Secrets Using DTrace on Oracle Solaris 11

Yu Wang
Principal Software Engineer, Oracle

Xiaosong Zhu
Principal Software Engineer, Oracle

Wen-Sheng Liu
Principal Software Engineer, Oracle

ORACLE

Table of Contents

Introduction

DTrace is a comprehensive dynamic tracing framework for the Oracle Solaris™ Operating System. DTrace provides a powerful infrastructure to troubleshoot kernel and application problems in production environment. It allows administrators and developers to concisely answer arbitrary questions about the behavior of user applications and operating systems.

JDK 8 is the most innovative version of Java ever. It brings many new features to the Java platform, such as Lambda Expressions, Streams, and Functional Interfaces. For the programmers, these features are easy to use; however, it is hard to understand their internal mechanisms.

In this lab, we will learn how to use the DTrace to find out how a Java Virtual Machine (JVM) implements its new features, including Streams Pipeline, Stream Parallelism, Lambda Lazy Evaluation, and Recursion Optimizing on Oracle Solaris 11.

Prerequisites

This hands-on lab assumes you have some basic knowledge about the following technologies.

- Java™ language programming
- Administration of Oracle Solaris or a similar UNIX or Linux OS

Hardware/Software Requirements

- Memory requirement: 8 GB
- Disk space requirement: 50 GB
- Operating System: Solaris 11.2
- Java SE Development Kit 8
- Gradle building tool

Environment Configuration

- Solaris 11.2 (with desktop packages) installed
- Add `<JDK_7_Installation>/bin` to your `PATH` variable.
- Add `<GRADLE_Installation>/bin` to your `PATH` variable
- Then go to Appendix A and—using the directory structure described there—use the `.java` files, and `build.gradle` files shown there to create the Java code, and Gradle build files used in the lab exercises.

Notes for Users

- The lab prefers the GNOME desktop environment over Oracle Solaris 11. (with desktop packages installed).
- In order to open a terminal window in GNOME, right-click any point on the background of the desktop, and select *Open Terminal* in the pop-up menu (as shown in Figure 1).

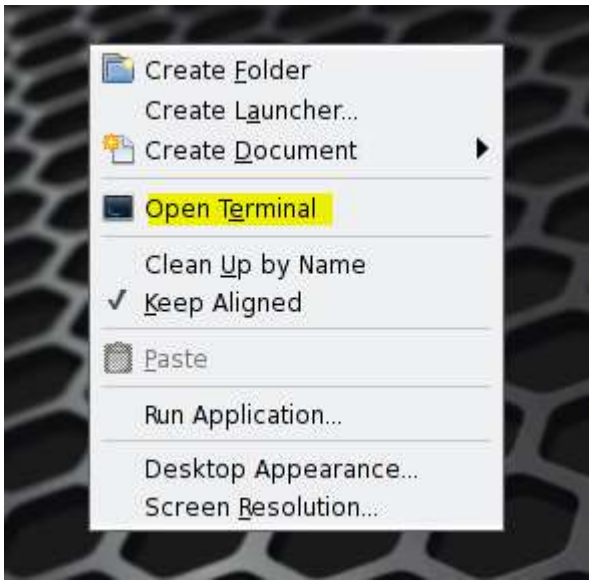

Figure 1. Open a terminal in Solaris 11

- DTrace requires root privileges to run. If you log into the desktop with a non-root user (as recommended) , you should first enable Dtrace for the user by a simple command:

```
# usermod -K defaultpriv=basic,dtrace_proc,dtrace_user,dtrace_kernel labuser
```

Note: This command has to be run as root. The user name "**labuser**" is used in this hands-on lab, you can use your own name as your choice.

Exercise 1: Introduction to DTrace (15 Minutes)

In this exercise we are going to learn some basic concepts about D-Language.

Background Information

What is DTrace?

DTrace is a comprehensive dynamic tracing facility that is built into Oracle Solaris that can be used by administrators and developers on live production systems to examine the behavior of both user programs and the operating system itself. DTrace enables you to explore your system to understand how it works, track down performance problems across many layers of software, or locate the cause of aberrant behavior. DTrace lets you create your own custom programs to dynamically instrument the system and provide immediate, concise answers to arbitrary questions you can formulate using the DTrace D programming language.

A D program source file consists of one or more probe clauses that describe the instrumentation to be enabled by DTrace. Each probe clause has the general form:

```
probe descriptions
/ predicates /
{
     action statements;
}
```

Step 1: 'Hello, world!'

1. After logging in, you may click the right mouse button on the desktop and choose Open Terminal to bring out a terminal window.

2.Type dtrace to print the different options of the DTrace command.

```
labuser$solaris11:~$ dtrace
Usage: dtrace [-32|-64] [-aACeFGhHlqSvVwZ] [-b bufsz] [-c cmd] [-D name[=def]]
    [-I path] [-L path] [-o output] [-p pid] [-s script] [-U name]
    [-x opt[=val]] [-X a|c|s|t]

    [-P provider [[ predicate ] action ]]
    [-m [ provider: ] module [[ predicate ] action ]]
    [-f [[ provider: ] module: ] func [[ predicate ] action ]]
    [-n [[[ provider: ] module: ] func: ] name [[ predicate ] action ]]
    [-i probe-id [[ predicate ] action ]] [ args ... ]

    predicate -> '/' D-expression '/'
       action -> '{' D-statements '}'

    -32 generate 32-bit D programs and ELF files
    -64 generate 64-bit D programs and ELF files

    -a  claim anonymous tracing state
    -A  generate driver.conf(4) directives for anonymous tracing
    -b  set trace buffer size
    -c  run specified command and exit upon its completion
    -C  run cpp(1) preprocessor on script files
```

```
    -D  define symbol when invoking preprocessor
    -e  exit after compiling request but prior to enabling probes
    -f  enable or list probes matching the specified function name
    -F  coalesce trace output by function
    -G  generate an ELF file containing embedded dtrace program
    -h  generate a header file with definitions for static probes
    -H  print included files when invoking preprocessor
    -i  enable or list probes matching the specified probe id
    -I  add include directory to preprocessor search path
    -l  list probes matching specified criteria
    -L  add library directory to library search path
    -m  enable or list probes matching the specified module name
    -n  enable or list probes matching the specified probe name
    -o  set output file
    -p  grab specified process-ID and cache its symbol tables
    -P  enable or list probes matching the specified provider name
    -q  set quiet mode (only output explicitly traced data)
    -s  enable or list probes according to the specified D script
    -S  print D compiler intermediate code
    -U  undefine symbol when invoking preprocessor
    -v  set verbose mode (report stability attributes, arguments)
    -V  report DTrace API version
    -w  permit destructive actions
    -x  enable or modify compiler and tracing options
    -X  specify ISO C conformance settings for preprocessor
    -Z  permit probe descriptions that match zero probes
```

4. Now let's try to write a "Hello, world!" by using D-Language:

```
labuser@solaris11:~$ dtrace -n 'BEGIN {trace("hello,world");exit(0);}'
dtrace: description 'BEGIN ' matched 1 probe
 CPU     ID                    FUNCTION:NAME
   0      1                           :BEGIN   hello,world
```

5.Alternatively, you can compose a script file, and saved as “ex1-hello.d”:

```
BEGIN
{
     trace("hello, world");
     exit(0);
}
```

Then run it:

```
labuser@solaris11:~$ dtrace -s ex1-hello.d
dtrace: script 'ex1-hello.d' matched 1 probe
 CPU     ID                    FUNCTION:NAME
   0      1                           :BEGIN   hello,world
```

Step 2: Probes

A probe is a location or activity to which DTrace can bind a request to perform a set of actions, like recording a stack trace, a timestamp, or the argument to a function. Probes are like programmable sensors scattered all over your Oracle Solaris system in interesting places. If you want to figure out what's going on, you can use DTrace to program the appropriate sensors to record the useful information.

1. What kinds of probes are there in Solaris 11.2? Let's see:

```
labuser@solaris11:~$ dtrace -l | more
   ID   PROVIDER            MODULE                              FUNCTION NAME
    1     dtrace                                                       BEGIN
    2     dtrace                                                       END
    3     dtrace                                                       ERROR
    4 javascript2010         libxul.so __1cCjsHExecute6FpnJJSContext_pnIJSScript_
rnIJSObject_pnCJSFValue__b_ execute-done
    5 javascript2010         libxul.so __1cCjsNExecuteKernel6FpnJJSContext_pnIJSS
cript_rnIJSObject_rknCJSFValue_n0ALExecuteType_pn0AKStackFrame_p8_b_ execute-done
    6 javascript2010         libxul.so __1cCjsHExecute6FpnJJSContext_pnIJSScript_
rnIJSObject_pnCJSFValue__b_ execute-start
    7 javascript2010         libxul.so __1cCjsNExecuteKernel6FpnJJSContext_pnIJSS
cript_rnIJSObject_rknCJSFValue_n0ALExecuteType_pn0AKStackFrame_p8_b_ execute-star
t
    8 nfsmapid1010          nfsmapid                  cb_update_domain daemon-dom
ain
    9 kerberos995    mech_krb5.so.1                       k5_mk_rep krb_ap_rep-
make
   10 kerberos995    mech_krb5.so.1                    krb5_rd_rep krb_ap_rep-
read
   11 kerberos995    mech_krb5.so.1             krb5_mk_req_extended krb_ap_req-
make
   12 kerberos995    mech_krb5.so.1               rd_req_decoded_opt krb_ap_req-
read
   13 kerberos995    mech_krb5.so.1                    krb5_mk_ncred krb_cred-ma
ke
   14 kerberos995    mech_krb5.so.1               krb5_rd_cred_basic krb_cred-re
ad
   15 kerberos995    mech_krb5.so.1                    krb5_mk_error krb_error-m
ake
   16 kerberos995    mech_krb5.so.1                    krb5_rd_error krb_error-r
ead
   17 kerberos995    mech_krb5.so.1              krb5_encode_kdc_rep krb_kdc_rep
-make
--More--
```

Every probe in DTrace has two names: a unique integer ID and a human-readable string name which is composed of four parts, shown as separate columns in the dtrace output. When writing out the full human-readable name of a probe, write all four parts of the name separated by colons like this:

```
provider:module:function:name
```

2. Type the following command to know how many probes are available.

```
labuser@solaris11:~$ dtrace -l | wc -l
   81183
```

Please note that the number of probes varies depending on your OS and the software you have installed.

3. In order to locate a DTrace probe, we can also use the following options to filter the list:

- -P for provider
- -m for module
- -f for function
- -n for name

For instance, if you want to list all of the probes from *syscall* provider

```
labuser@solaris11:~$ dtrace -l -P syscall | more
   ID   PROVIDER            MODULE                              FUNCTION NAME
 7067     syscall                                                 nosys entry
 7068     syscall                                                 nosys return
 7069     syscall                                                 rexit entry
```

```
 7070    syscall                                                  rexit return
 7071    syscall                                                   read entry
 7072    syscall                                                   read return
 7073    syscall                                                  write entry
 7074    syscall                                                  write return
 7075    syscall                                                  close entry
 7076    syscall                                                  close return
 7077    syscall                                                 linkat entry
 7078    syscall                                                 linkat return
 7079    syscall                                               symlinkat entry
 7080    syscall                                               symlinkat return
 7081    syscall                                                   chdir entry
 7082    syscall                                                   chdir return
 7083    syscall                                                   gtime entry
 7084    syscall                                                   gtime return
 7085    syscall                                                     brk entry
 7086    syscall                                                     brk return
 7087    syscall                                                   lseek entry
--More--
```

Step 3: Predicates and Actions

Predicates are expressions enclosed in slashes / / that are evaluated at probe firing time to determine whether the associated actions should be executed. Predicates are the primary conditional construct used for building more complex control flow in a D program. You can omit the predicate section of the probe clause entirely for any probe, in which case the actions are always executed when the probe fires.

Probe actions are described by a list of statements separated by semicolons (;) and enclosed in braces { }. If you only want to note that a particular probe fired on a particular CPU without tracing any data or performing any additional actions, you can specify an empty set of braces with no statements inside.

1. Now you will write a script to list all of system calls, excluding *write*, for all the processes running on your current system. Save this file as "**ex1-predicate-1.d"**

```
syscall:::
/probefunc!="write"/
{
          printf("probefunc:%s, pid:%d, execname:%s\n",probefunc, pid, execname);
}
```

2. Run **ex1-predicate-1.d**. You may get the similar result as below. Press **Ctrl-c** to stop this program

```
labuser@solaris11:~$ dtrace -qs ex1-predicate-1.d
^C
probefunc:ioctl, pid:2084, execname:dtrace
probefunc:ioctl, pid:2084, execname:dtrace
probefunc:ioctl, pid:2084, execname:dtrace
probefunc:ioctl, pid:2084, execname:dtrace
probefunc:portfs, pid:1829, execname:nautilus
probefunc:portfs, pid:1829, execname:nautilus
probefunc:portfs, pid:1829, execname:nautilus
probefunc:portfs, pid:1829, execname:nautilus
probefunc:fstatat64, pid:1829, execname:nautilus
probefunc:fstatat64, pid:1829, execname:nautilus
probefunc:clock_gettime, pid:1829, execname:nautilus
probefunc:clock_gettime, pid:1829, execname:nautilus
probefunc:recv, pid:1829, execname:nautilus
probefunc:recv, pid:1829, execname:nautilus
probefunc:clock_gettime, pid:1829, execname:nautilus
probefunc:clock_gettime, pid:1829, execname:nautilus
probefunc:pollsys, pid:1829, execname:nautilus
probefunc:mmap, pid:2084, execname:dtrace
probefunc:mmap, pid:2084, execname:dtrace
```

```
probefunc:lwp_park, pid:2084, execname:dtrace
probefunc:recv, pid:1871, execname:VBoxClient
probefunc:recv, pid:1871, execname:VBoxClient
probefunc:nanosleep, pid:1871, execname:VBoxClient
probefunc:lwp_cond_wait, pid:1844, execname:java
probefunc:nanosleep, pid:1871, execname:VBoxClient
probefunc:recv, pid:1871, execname:VBoxClient
probefunc:recv, pid:1871, execname:VBoxClient
probefunc:nanosleep, pid:1871, execname:VBoxClient
probefunc:lwp_cond_wait, pid:1844, execname:java
probefunc:lwp_cond_signal, pid:1844, execname:java
probefunc:lwp_cond_signal, pid:1844, execname:java
probefunc:lwp_cond_wait, pid:1844, execname:java
probefunc:lwp_cond_wait, pid:1844, execname:java
probefunc:lwp_cond_signal, pid:1844, execname:java
probefunc:lwp_cond_signal, pid:1844, execname:java
```

3. In order to find out all of no-write system calls caused by java, we need to make small changes as below and save this file as "ex1-predicate-2.d"

```
syscall:::
/probefunc!="write" && execname=="java"/
{
          printf("probefunc:%s, pid:%d, execname:%s\n",probefunc, pid, execname);
}
```

4. Run **ex1-predicate-2.d**. You will get the result like this:

```
labuser@solaris11:~$ dtrace -qs ex1-predicate-2.d
^C
probefunc:lwp_cond_wait, pid:1844, execname:java
probefunc:lwp_cond_wait, pid:1844, execname:java
probefunc:lwp_cond_signal, pid:1844, execname:java
probefunc:lwp_cond_signal, pid:1844, execname:java
probefunc:lwp_cond_wait, pid:1844, execname:java
probefunc:lwp_cond_wait, pid:1844, execname:java
probefunc:lwp_cond_signal, pid:1844, execname:java
probefunc:lwp_cond_signal, pid:1844, execname:java
probefunc:lwp_cond_wait, pid:1844, execname:java
probefunc:lwp_cond_wait, pid:1844, execname:java
probefunc:lwp_cond_wait, pid:1844, execname:java
probefunc:lwp_cond_wait, pid:1844, execname:java
probefunc:lwp_cond_wait, pid:1844, execname:java
probefunc:lwp_cond_wait, pid:1844, execname:java
probefunc:lwp_cond_signal, pid:1844, execname:java
probefunc:lwp_cond_signal, pid:1844, execname:java
probefunc:lwp_cond_wait, pid:1844, execname:java
probefunc:lwp_cond_wait, pid:1844, execname:java
probefunc:lwp_cond_signal, pid:1844, execname:java
```

Step 4: Aggregations

When instrumenting the system to answer performance-related questions, it is useful to consider how data can be aggregated to answer a specific question rather than thinking in terms of data gathered by individual probes. For example, if you wanted to know the number of system calls by user ID, you would not necessarily care about the datum collected at each system call. You simply want to see a table of user IDs and system calls. In DTrace, you will find it is very easy to achieve this by using aggregations.

In D-Language, the syntax for an aggregation is:

```
@name[ keys ] = aggfunc ( args );
```

- name: the name of the aggregation
- keys: a comma-separated list of D expressions
- aggfunc: a DTrace aggregating function
- args: a comma-separated list of arguments appropriate for the aggregating function

1. In this example, we'd like to know how many system calls are invoked within 5 seconds. Write some scripts as following and save it to the file named "**ex1-aggregation-1.d**"

```
syscall:::entry
{
   @counts["syscall numbers"]=count();
}
tick-5sec
{
   exit(0);
}
```

2. Run **ex1-aggregation-1.d** which will exit in 5 seconds, you will see the similar result like this:

```
labuser@solaris11:~$ dtrace -s ex1-aggregation-1.d
dtrace: script 'ex1-aggregation-1.d' matched 215 probes
 CPU     ID                    FUNCTION:NAME
   0  81183                       :tick-5sec

  syscall numbers                                                3420
```

3.Take a further step to check which application is making the most system calls. Then we need to update the original file as below and save it to the **ex1-aggregation-2.d** file.

```
syscall:::entry
{
          @counts[execname]=count();
}
tick-5sec
{
          exit(0);
}
```

4. Running **ex1-aggregation-2.d** file again will bring you the result like this:

```
labuser@solaris11:~$ dtrace -s ex1-aggregation-2.d
dtrace: script 'ex1-aggregation-2.d' matched 215 probes
 CPU     ID                    FUNCTION:NAME
   0  81183                       :tick-5sec

  automountd                                                        1
  dhcpagent                                                         3
  nwam-manager                                                      4
  devfsadm                                                          6
  gnome-settings-d                                                  7
  isapython2.6                                                      8
  sendmail                                                         10
  metacity                                                         11
  xscreensaver                                                     12
  gnome-power-mana                                                 15
  updatemanagernot                                                 16
  VBoxService                                                      37
  VBoxClient                                                      354
  gnome-terminal                                                  361
  Xorg                                                            393
  java                                                            596
  dtrace                                                         1573
```

5. The following example displays the average time spent in the system calls, organized by process name. This example uses the avg aggregating function, specifying *timestamp - self->ts* as the argument. The example averages the wall clock time spent in the system call. Please save the scripts to the file named "**ex1-aggregation-3.d".**

```
syscall:::entry
{
          self->ts=timestamp;
}
syscall:::return
/self->ts/
{
          @time[execname]=avg(timestamp-self->ts);
          self->ts=0;
}
```

About *self->*: Thread-Local Variables

DTrace provides the ability to declare variable storage that is local to each operating system thread, as well as the global variables mentioned previously. Thread-local variables are useful when we want to enable a probe and mark every thread that fires the probe with some tags or data. In a D program we can use thread-local variables to share a common name but refer to different data storage associated with different threads. Thread-local variables are referenced by applying the -> operator to the special identifier self.

6. While running **ex1-aggregation-3.d** file, you need to wait for several seconds, then press **Ctrl-c**. The similar result will be shown:

```
labuser@solaris11:~$ dtrace -s ex1-aggregation-3.d
dtrace: script 'ex1-aggregation-3.d' matched 428 probes
^C

  fmd                                                           18894
  nscd                                                          25965
  utmpd                                                         28892
  ssh-agent                                                     37801
  gconfd-2                                                      79348
  nwam-manager                                                  79479
  hald                                                          88817
  hald-addon-acpi                                               95590
  dtrace                                                      3307590
  VBoxClient                                                 13981578
  Xorg                                                       15435316
  gnome-terminal                                             20763581
  java                                                       40776641
  gnome-power-mana                                           58499707
  xscreensaver                                              184487447
  updatemanagernot                                          241129100
  sendmail                                                  280901476
  VBoxService                                               358442957
  gnome-settings-d                                          392415577
  metacity                                                  409615593
  isapython2.6                                              502919529
  dhcpagent                                                 800092245
  devfsadm                                                  914232179
```

The above average time records are displayed in nanoseconds.

7. If you want to know the distribution of system call duration from different applications, you need to use quantize aggregation function as following. Please save the scripts to the file named **ex1-aggregation-4.d** :

```
syscall:::entry
{
          self->ts=timestamp;
}
syscall:::return
/self->ts/
{
          @time[execname]=quantize(timestamp-self->ts);
```

```
        self->ts=0;
}
```

8. Run **ex1-aggregation-4.d** again and wait for several seconds. You should get the result like this:

```
labuser@solaris11:~$ dtrace -s ex1-aggregation-4.d
dtrace: script 'ex1-aggregation-4.d' matched 428 probes
^C

  nwam-manager
           value  ------------- Distribution ------------- count
           16384 |                                         0
           32768 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 3
           65536 |                                         0

  sendmail
           value  ------------- Distribution ------------- count
             512 |                                         0
            1024 |@@@@                                     1
            2048 |@@@@@@@@@@@@@@                           3
            4096 |                                         0
            8192 |@@@@                                     1
           16384 |@@@@                                     1
           32768 |@@@@@@@@@@@@@@                           3
           65536 |                                         0

  gnome-power-mana
           value  ------------- Distribution ------------- count
            1024 |                                         0
            2048 |@@@@@@@@@@@@                             4
            4096 |@@@@@@@@@@                               3
            8192 |@@@@@@@@@@                               3
           16384 |                                         0
           32768 |                                         0
           65536 |                                         0
          131072 |@@@                                      1
          262144 |@@@                                      1
          524288 |                                         0
         1048576 |@@@                                      1
         2097152 |                                         0
         4194304 |                                         0
         8388608 |                                         0
        16777216 |                                         0
        33554432 |                                         0
        67108864 |                                         0
       134217728 |                                         0
       268435456 |@@@                                      1
       536870912 |                                         0
...
```

Note that the rows for the frequency distribution are always power-of-two values. Each rows indicates the count of the number of elements greater than or equal to the corresponding value, but less than the next larger row value. For example, the above output shows that sendmail had 3 system calls taking between 2,048 nanoseconds and 4,096 nanoseconds, inclusive.

Summary

In this exercise, you have learned the basic structure of a DTrace program. Also you just wrote some simple D programs to trace the system and application information

Exercise 2: Observing Lazy Evaluation with DTrace (10 Minutes)

JVM often executes code eagerly. The arguments are evaluated right at the time of method calls. While in Java 8, we could construct Lambda expressions which can make it evaluated lazily. In this exercise, we will use a "hotspot" DTrace provider, which is available in the Java SE 8 HotSpot VM on Solaris, to trace the Java method calls and identify the lazy evaluation behavior.

What is hotspot Provider?

With the introduction of DTrace into Solaris, DTrace support has been added to the Java SE 8 HotSpot VM. The *hotspot* provider makes available probes that can be used to monitor JVM internal state and activities as well as the Java application that is running. All of the probes are USDT probes and are accessed using the process-id of the JVM process.

The *hotspot* provider makes available probes that can be used to track the lifespan of the VM, thread start and stop events, GC and memory pool statistics, method compilations, and monitor activity. With a startup flag, additional probes are enabled which can be used to monitor the running Java program, such as method enter and return probes, and object allocations. All of the *hotspot* probes originate in the VM library (**libjvm.so**), so they are also provided from programs which embed the VM.

Background Information

Check the following Java Class, which is validating a driver license by the driver's age and the license identity code. Suppose checking the age is easy and fast (with the method of "*isOldEnough()*") and checking the license's code is hard and slow (with the method of *isDriverLicenseValid()*). We have two versions of methods to validate of the license, one is formal way, show as the method of "*isDriverLegal1()*", the other is constructed with Lambda expressions, show as the method of "*isDriverLegal2()*" . The logic is the same: first we check the age, if the driver is old enough, then we check the license code. Ideally, if we find the driver is not old enough, we should not check the license code and return false quickly.

The java source file (*LicenseEvaluate.java*) is located in **<lab_root>/OOWHOL10194/Applications src/main//java/Exercises/exercise2/**

```
package Exercises.exercise1;
import java.util.function.Supplier;

public class LicenseEvaluate {

    public static boolean isOldEnough(int age){
        return age >= 18 ;
    }
    public static boolean isDriverLicenseValid (final int licensecode) {
        comsumingProcess();
        return true;
    }

    private static boolean isDriverLegal1(final boolean input1,final boolean input2){
        return input1&&input2;
    }

    private static boolean isDriverLegal2(final Supplier<Boolean> input1,final Supplier<Boolean> input2){
        return input1.get() && input2.get();
    }


    public static void eagerEvaluate(final int age, final int code){
```

```
        boolean result = isDriverLegal1(isOldEnough(age),isDriverLicenseValid(code));
        if (result) System.out.println("It's legal!");
        else System.out.println("It is illegal!");
    }


    public static void lazyEvaluate(final int age, final int code){
        boolean result = isDriverLegal2(() -> isOldEnough(age), () -> isDriverLicenseValid(code));
        if (result) System.out.println("It's legal!");
        else System.out.println("It is illegal!");
    }


    private static void comsumingProcess() {
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Step 1: Run the application of Exercise 2

1. Click the right mouse button on the desktop and choose *Open Terminal* to bring up a terminal window.

2. Go to the directory of **<lab_root>/OOWHOL6427/Applications/**, execute the command of **gradle -q** as following:

```
labuser@Solaris11:~$  cd OOWHOL6427/Applications
labuser@Solaris11:~/OOWHOL6427$  gradle -q

Eager            :  Exercise2---Run EagerEvaluation
Lazy             :  Exercise2---Run LazyEvaluation
Pipeline         :  Exercise3---Run StreamPipeline
RecursiveCall    :  Exercise4---Run RecursiveCall
OpRecursiveCall  :  Exercise4---Run Optimized RecursiveCall
NonParallel      :  Exercise5---Run Non-Parallel Stream
Parallel         :  Exercise5---Run Parallel Stream

Print the your choice please:
>
```

This **gradle** command runs a console program, which is interactive through the terminal. You can type the application name to choose the exercise you want to run. You can print the help information by press **Enter** key at any time, and stop it by enter the **"stop"** keyword.

3. Run the Exercise2 application with the input "eager" and "lazy". You will find the result out is the same, but the lazy one is faster.

```
Eager            :  Exercise2---Run EagerEvaluation
Lazy             :  Exercise2---Run LazyEvaluation
Pipeline         :  Exercise3---Run StreamPipeline
Serial           :  Exercise4---Run Serial Stream
Parallel         :  Exercise4---Run Parallel Stream
RecursiveCall    :  Exercise5---Run RecursiveCall
OpRecursiveCall  :  Exercise5---Run Optimized RecursiveCall

Print the your choice please:

>eager
It is illegal!

>lazy
It is illegal!
```

Step 2: Monitoring the program with JDK tools

1. Open another terminal, and run the "jps" command as following:

```
labuser@solaris11:~# jps
1701 GradleMain
1717 OOW2015Hol6427
1742 Jps
labuser@solaris11:~#
```

By using the **jps**, we can find each Java processes with its process id. From the output above, We can tell that the process id of the above Java program on my machine is 1717(<pid>=1717, maybe different on your machine), for it has a name of *OOW2015Hol6427*. Then we can pick it out from other Java processes.

Step 3: Writing a DTrace script to identify the laziness of Lambda expression

1. We will use "hotspot provider" to trace Java method calls to identify the reason why the lazy method is faster than the eager one.

About Hotspot Provider and Application Tracing Probes

All of the probes in hotspot provider are USDT probes and are accessed using the process-id of the JVM process. A Java process with an ID of 123, for example, would be traced by using the "hotspot123" provider. Usually we use a variable such as $target or $1 as a parameter to define the provider dynamically.

A few probes are provided to allow fine-grained examination of Java thread execution. These consist of probes that fire anytime a method is entered or returned from, as well as a probe that fires whenever a Java object has been allocated.

Method-entry probe: Probe that fires when a method is being entered.
method-return probe: Probe that fires when a method returns, either normally or due to an exception

Method probe arguments:
arg0: The Java thread ID of the thread that is entering or leaving the method
arg1: A pointer to UTF-8 string data which contains the name of the class of the method
arg2: The length of the class name data (in bytes)
arg3: A pointer to UTF-8 string data which contains the name of the method
arg4: The length of the method name data (in bytes)
arg5: A pointer to UTF-8 string data which contains the signature of the method
arg6: The length of the signature data (in bytes)

In this exercise, we will use the following DTrace script to trace Java method calls. Shown as below, the script**<lab_root>/OOWHOL6427/DScripts/ex2.d** is to trace methods call we interested in:

```
#!/usr/sbin/dtrace -qs

hotspot$target:::method-entry
/(stringof(copyin(arg1, 9)) == "Exercises")/
{
printf("-> %4s.%s\n",
          stringof(copyin(arg1, arg2)), stringof(copyin(arg3, arg4)));
}

hotspot$target:::method-return
{}
```

Note: this script is to display the class and method name when it is being called in the designated java process of interest. The process id is designated by the $target macro variable (which will be often referred in our lab.). The name arg1 and arg3 is the class and method string, while arg2, arg4 is the length of the string. For we know that DTrace is running in kernel mode, when tracing the user processes (in this case, it is java process), we need to use "copyin" function to copy the class/method string from user address space into kernel. To prevent the unrelated output, we use the predicate `(stringof(copyin(arg1, 9)) == "Exercises")` to display only the class that is related with our lab exercises.

2. Open another terminal window, go to the directory "**<lab_root>/OOWHOL6427/DScripts**" and run the "ex2.d" script as following. (the number "1717" is the process id got from "jps" in the last step, your id maybe diferrent.)

```
labuser@Solaris11:~$  cd OOWHOL6427/DScripts/
labuser@Solaris11:~/OOWHOL6427/DScripts$ ./ex2.d -p 1717
```

3. In the application terminal window, type "eager", and press **Enter** key, you will run the "eager evaluation" process, just as following:

```
Eager            :  Exercise2---Run EagerEvaluation
Lazy             :  Exercise2---Run LazyEvaluation
Pipeline         :  Exercise3---Run StreamPipeline
Serial           :  Exercise4---Run Serial Stream
Parallel         :  Exercise4---Run Parallel Stream
RecursiveCall    :  Exercise5---Run RecursiveCall
OpRecursiveCall  :  Exercise5---Run Optimized RecursiveCall

Print the your choice please:
>eager
It is illegal!
```

4. In the DTrace script window, you will find some output, just like following:

```
-> Exercises/exercise1/LicenseEvaluate.eagerEvaluate
-> Exercises/exercise1/LicenseEvaluate.isOldEnough
-> Exercises/exercise1/LicenseEvaluate.isDriverLicenseValid
-> Exercises/exercise1/LicenseEvaluate.consumingProcess
-> Exercises/exercise1/LicenseEvaluate.isDriverLegal1
```

The output shows that the "eager evaluate" process calls the "`isDriverLicenseValid`" the method, even the method "`isOldEnough`" return false.

5. In the application terminal window, type "lazy", and press **Enter** key, you will run the "lazy evaluation" process, just as following:

```
labuser@Solaris11:~/OOWHOL6427$  gradle -q

Eager            :  Exercise2---Run EagerEvaluation
Lazy             :  Exercise2---Run LazyEvaluation
Pipeline         :  Exercise3---Run StreamPipeline
Serial           :  Exercise4---Run Serial Stream
Parallel         :  Exercise4---Run Parallel Stream
RecursiveCall    :  Exercise5---Run RecursiveCall
OpRecursiveCall  :  Exercise5---Run Optimized RecursiveCall

Print the your choice please:
>eager
I am here....
It is illegal!
>lazy
It is illegal!
```

6. In the DTrace script window, you will find some output, just like following:

```
-> Exercises/exercise1/LicenseEvaluate.lazyEvaluate
-> Exercises/exercise1/LicenseEvaluate$$Lambda$1.get$Lambda
-> Exercises/exercise1/LicenseEvaluate$$Lambda$1.<init>
-> Exercises/exercise1/LicenseEvaluate$$Lambda$2.get$Lambda
-> Exercises/exercise1/LicenseEvaluate$$Lambda$2.<init>
-> Exercises/exercise1/LicenseEvaluate.isDriverLegal2
```

```
-> Exercises/exercise1/LicenseEvaluate$$Lambda$1.get
-> Exercises/exercise1/LicenseEvaluate.lambda$lazyEvaluate$0
-> Exercises/exercise1/LicenseEvaluate.isOldEnough
```

The output shows that the "lazy evaluate" process DONOT make a call to the "`isDriverLicenseValid`" method and the "`consumingProcess`" method. The lazy evaluation process skips them when the method "`isOldEnough`" returns false.

7. In the DTrace script window, type **CTRL+C** to stop the script.

Summary

In this exercise, you have created and run a DTrace script to trace the Java method calls. By observing the output of the DTrace script, we find that Lambda expressions in Java 8 can really bring lazy evaluation, and it can improve application performance under some circumstances.

Exercise 3: Observing Stream Pipeline with DTrace (10 Minutes)

The most important changes in JDK 8 are focused on the Collections API and its new addition: Streams. Streams allow us to write collections-processing code at a higher level of abstraction. While the codes with Streams in JDK 8 are more expressive and more concise, the beginners are sometimes confusing about the execution of the Streams underlined. In this exercise, we will use DTrace script to explore the Stream pipeline when it is running.

Background Information

Check the following Java Classes, "Movie" and "StreamPipeline". The codes are very straight forward. The Movie class has some static methods, to judge whether a movie is popular (if the score is larger than 9.6), whether a movie is recent (if the year is larger than 2008). Also the Movie class has a static method to change its name to upper cases. The StreamPipeline class is generating a stream from a list of Movies. Then it filters the list and gets only recent movies, and then filters it again to get only popular ones. After these operations, the movies' names are changed to upper case. At last, we get the first movie that satisfied all the filters and print its name on the screen.

What is the pipeline of this program when it is running? Will the first filter go through all the movies in the list to find the recent ones and then pass those to the second filter to find the popular ones? If it is true, we can get a conclusion from the source codes that the program will call the "`isRecent`" method 12 times, call the "`isVeryPopular`" method 7 times and call the "`getUpperName`" method 7 times too. We will write a DTrace script to verify it.

The java source files (StreamPipeline.java and Movie.java) are located in **<lab_root>/OOWHOL10194/ Applications/src/main//java/Exercises/exercise3/**

```
public class Movie {

    private String name;
    private int releaseYear;
    private double score;
    public Movie(String m_name,int m_year,double m_score) {
        name = m_name;
        releaseYear = m_year;
        score = m_score;
    }

    public static boolean isVeryPopular(Movie movie) {
        return movie.score>9.6;
    }

    public static boolean isRecent(Movie movie) {
        return movie.releaseYear>2008;
    }

    public static String getUppperName(Movie movie) {
        return movie.name.toUpperCase();
    }

    public static List<Movie> getPopularMovies() {
        List<Movie> popularMovies = new ArrayList<Movie>();
        popularMovies.add(new Movie("Boyhood",2014,10.0));
        popularMovies.add(new Movie("Pan's Labyrinth",2006,9.8));
        popularMovies.add(new Movie("Carol",2015,9.8));
        popularMovies.add(new Movie("Hoop Dreams",1994,9.8));
        popularMovies.add(new Movie("4 Months, 3 Weeks and 2 Days",2008,9.7));
        popularMovies.add(new Movie("12 Years A Slave",2013,9.7));
        popularMovies.add(new Movie("Dr. Strangelove",1964,9.7));
        popularMovies.add(new Movie("Ratatouille",2007,9.6));
```

```
        popularMovies.add(new Movie("Gravity",2013,9.6));
        popularMovies.add(new Movie("The Social Network",2010,9.5));
        popularMovies.add(new Movie("Zero Dark Thirty",2012,9.5));
        popularMovies.add(new Movie("Zero Dark Thirty",2011,9.5));
        return popularMovies;
    }
}
```

```
public class StreamPipeline {
    public static void getMyLuckyMovieNames() {
        List<Movie> popularMovies = Movie.getPopularMovies();
        String myLuckyMovie = popularMovies.stream()
                .filter(Movie::isRecent)
                .filter(Movie::isVeryPopular)
                .map(Movie::getUppperName)
                .findFirst()
                .get();

        System.out.println("I found a good movie: "+myLuckyMovie);
    }
}
```

Step 1: Run the application of Exercise 3

(if the application terminal window is still open after exercise 2, you can reuse it, just skip to item 3.)

1. If the application is closed after the last exercise or you skip last exercises, please click the right mouse button on the desktop and choose *Open Terminal* to bring up a terminal window.

2. Go to the directory of **<lab_root>/OOWHOL6427/Applications**, execute the command of **gradle -q** as following:

```
labuser@Solaris11:~$  cd OOWHOL6427/Applications
labuser@Solaris11:~/OOWHOL6427$  gradle -q

Eager            :  Exercise2---Run EagerEvaluation
Lazy             :  Exercise2---Run LazyEvaluation
Pipeline         :  Exercise3---Run StreamPipeline
Serial           :  Exercise4---Run Serial Stream
Parallel         :  Exercise4---Run Parallel Stream
RecursiveCall    :  Exercise5---Run RecursiveCall
OpRecursiveCall  :  Exercise5---Run Optimized RecursiveCall

Print the your choice please:
>
```

This **gradle** command runs a console program, which is interactive through the terminal. You can type the application name to choose the exercise you want to run. You can print the help information by press **Enter** key at any time, and stop it by enter the **"stop"** keyword.

3. Run the Exercise2 application with the input "pipeline". You will it will print a movie named "BOYHOOD".

```
Eager            :  Exercise2---Run EagerEvaluation
Lazy             :  Exercise2---Run LazyEvaluation
Pipeline         :  Exercise3---Run StreamPipeline
Serial           :  Exercise4---Run Serial Stream
Parallel         :  Exercise4---Run Parallel Stream
RecursiveCall    :  Exercise5---Run RecursiveCall
OpRecursiveCall  :  Exercise5---Run Optimized RecursiveCall

Print the your choice please:

>pipeline
I found a good movie:BOYHOOD
```

Step 2: Monitoring the program with JDK tools

1. Open another terminal, and run the "jps" command as following:

```
labuser@solaris11:~# jps
1701 GradleMain
1717 OOW2015Hol6427
1742 Jps
labuser@solaris11:~#
```

By using the **jps**, we can find each Java processes with its process id. From the output above, We can tell that the process id of the above Java program on my machine is 1717(<pid>=1717, maybe different on your machine), for it has a name of *OOW2015Hol6427*. Then we can pick it out from other Java processes.

Step 3: Writing a DTrace script to observe the stream pipeline

1. We will use "hotspot provider" to trace Java method calls to identify how many times these methods are called.

In this exercise, we will use the following DTrace script to trace Java method calls. Shown as below, the script**<lab_root>/OOWHOL6427/DScripts/ex3.d** is to trace methods call we interested in:

```
#!/usr/sbin/dtrace -qs

dtrace:::BEGIN
{
          printf("Tracing... Hit Ctrl-C to end.\n");
}

hotspot$target:::method-entry
/(stringof(copyin(arg1, 9)) == "Exercises")/
{
          class = (char *) copyin(arg1, arg2 + 1);
          class[arg2] = '\0';
          method = (char *) copyin(arg3, arg4 + 1);
          method[arg4] = '\0';
          @calls[stringof(class), stringof(method)] = count();
}

dtrace:::END
{
          printa("%48s.%-24s %@4d\n", @calls);
}
```

Note: When any method of our interested classes(class name includes Exercises) is called, the script will record the calling times using aggregate function count(), and store it in @calls variable, and sort by class name and method name.

2. Open another terminal window, go to the directory "**<lab_root>/OOWHOL6427/DScripts**" and run the "ex3.d" script as following. (the number "1717" is the process id got from "jps" in the last step, your id maybe diferrent.)

```
labuser@Solaris11:~$ cd OOWHOL6427/DScripts/
labuser@Solaris11:~/OOWHOL6427/DScripts$ ./ex3.d -p 1717
Tracing... Hit Ctrl-C to end.
```

3. In the application terminal window, type "pipeline" again, and press **Enter** key, you will run the "pipeline" process, just as following:

```
Eager            :  Exercise2---Run EagerEvaluation
Lazy             :  Exercise2---Run LazyEvaluation
Pipeline         :  Exercise3---Run StreamPipeline
Serial           :  Exercise4---Run Serial Stream
Parallel         :  Exercise4---Run Parallel Stream
RecursiveCall    :  Exercise5---Run RecursiveCall
```

```
OpRecursiveCall  :  Exercise5---Run Optimized RecursiveCall
Print the your choice please:
>pipeline
I found a good movie:BOYHOOD
```

4. In the DTrace script window, type the CTL+C, you will find some output, just like following:

```
Tracing... Hit Ctrl-C to end.
^C
                        Exercises/exercise2/Movie.getPopularMovies            1
                        Exercises/exercise2/Movie.getUppperName               1
                        Exercises/exercise2/Movie.isRecent                    1
                        Exercises/exercise2/Movie.isVeryPoular                1
              Exercises/exercise2/StreamPipeline.getMyLuckyMovieNames         1
    Exercises/exercise2/StreamPipeline$$Lambda$3.test                         1
    Exercises/exercise2/StreamPipeline$$Lambda$4.test                         1
    Exercises/exercise2/StreamPipeline$$Lambda$5.apply                        1
                        Exercises/exercise2/Movie.<init>                     12
```

To your surprise, the output shows that the methods of "isRecent" and "isVeryPopular" are called only once. It shows the stream DOESNOT execute the filter one by one and pass through all the items in the stream. Instead, the stream is very clever, it combines all the filters processing together, and it will stop immediately when it finds what it wants.

Summary

In this exercise, you have created and run a DTrace script to trace the Java method calls and count the times of every interested methods calling. By observing the output of the DTrace script, we find that Stream pipeline executes the filters' processing very efficiently and cleverly. When the items in the list pass through, the Stream will combine all operations together to make processing more efficiently and stop immediately when it finds what it wants to save resources and improve performance.

Exercise 4: Observing Stream Parallelism with DTrace (10 Minutes)

Parallel computing involves dividing a problem into sub-problems, solving those problems simultaneously (in parallel, with each sub-problem running in a separate thread), and then combining the results of the solutions to the sub-problems. Many developers think that streams are the most valuable feature in JDK 8, because they believe that by changing a single word in their programs
("replacing" stream with "parallelStream") they will make these programs work in parallel.

In this exercise, we will use DTrace to find out how the Java runtime partitions the stream into multiple parts of work when a stream executes in parallel, and help to understand what is really happening under the hood of stream parallelism.

Background Information

Check the following Java Class, "`StreamParallel`". The codes are very straight forward. We have bunch of transactions, and two versions of processing methods. One is the "`serialProcess`" method; the other is "`ParallelProcess`" method. The total difference between these two methods is just additional keyword "parallel" in the "`ParallelProcess`" method.

We will write a DTrace script to find how different it brings with just a single word!

The java source files (StreamParallel.java and other files) are located in **<lab_root>/OOWHOL6427/ Applications/src/main//java/Exercises/exercise4/**

```
public class StreamParallel {
    static List<Transaction> transactions = new ArrayList<Transaction>();
    static {
        transactions.add(new Transaction("one"));
        transactions.add(new Transaction("two"));
        transactions.add(new Transaction("three"));
        transactions.add(new Transaction("four"));
        transactions.add(new Transaction("five"));
        transactions.add(new Transaction("six"));
        transactions.add(new Transaction("seven"));
        transactions.add(new Transaction("eight"));
    }

    public static void ParallelProcess(){
        transactions.stream().parallel().forEach(t -> t.process());
    }

    public static void serialProcess(){
        transactions.stream().forEach(t -> t.process());
    }
}
```

Step 1: Run the application of Exercise 4

(if the application terminal window is still open after exercise 3, you can reuse it, just skip to item 3.)
1. If the application is closed after the last exercise or you skip exercises, please click the right mouse button on the desktop and choose *Open Terminal* to bring up a terminal window.

2. Go to the directory of **<lab_root>/OOWHOL6427/Applications**, execute the command of **gradle -q** as following:

```
labuser@Solaris11:~$  cd OOWHOL6427/Applications
labuser@Solaris11:~/OOWHOL6427$  gradle -q

Eager            :  Exercise2---Run EagerEvaluation
Lazy             :  Exercise2---Run LazyEvaluation
Pipeline         :  Exercise3---Run StreamPipeline
Serial           :  Exercise4---Run Serial Stream
Parallel         :  Exercise4---Run Parallel Stream
RecursiveCall    :  Exercise5---Run RecursiveCall
OpRecursiveCall  :  Exercise5---Run Optimized RecursiveCall

Print the your choice please:
>
```

This **gradle** command runs a console program, which is interactive through the terminal. You can type the application name to choose the exercise you want to run. You can print the help information by press **Enter** key at any time, and stop it by enter the **"stop"** keyword.

3. Run the Exercise4 "Serial Stream" application with the input "serial". You will find out that the transactions are executed one by one in perfect order.

```
Eager            :  Exercise2---Run EagerEvaluation
Lazy             :  Exercise2---Run LazyEvaluation
Pipeline         :  Exercise3---Run StreamPipeline
Serial           :  Exercise4---Run Serial Stream
Parallel         :  Exercise4---Run Parallel Stream
RecursiveCall    :  Exercise5---Run RecursiveCall
OpRecursiveCall  :  Exercise5---Run Optimized RecursiveCall
Print the your choice please:

>serial
working on one.....
working on two.....
working on three.....
working on four.....
working on five.....
working on six.....
working on seven.....
working on eight.....
>
```

4. Run the Exercise4 "Parallel Stream" application with the input "parallel". You will find that the transactions are executed two (or four) at a time and without any order.

```
Eager            :  Exercise2---Run EagerEvaluation
Lazy             :  Exercise2---Run LazyEvaluation
Pipeline         :  Exercise3---Run StreamPipeline
Serial           :  Exercise4---Run Serial Stream
Parallel         :  Exercise4---Run Parallel Stream
RecursiveCall    :  Exercise5---Run RecursiveCall
OpRecursiveCall  :  Exercise5---Run Optimized RecursiveCall
Print the your choice please:

>parallel
working on five.....
working on three.....
working on four.....
working on six.....
working on one.....
working on seven.....
```

```
working on eight.....
working on two.....
>
```

Step 2: Monitoring the program with JDK tools

1. Open another terminal, and run the "jps" command as following:

```
labuser@solaris11:~# jps
1701 GradleMain
1717 OOW2015Hol6427
1742 Jps
labuser@solaris11:~#
```

By using the **jps**, we can find each Java processes with its process id. From the output above, We can tell that the process id of the above Java program on my machine is 1717(<pid>=1717, maybe different on your machine), for it has a name of *OOW2015Hol6427*. Then we can pick it out from other Java processes.

Step 3: Writing a DTrace script to observe the multiple threads behavior

1.In this exercise, we will use the following DTrace script to trace the JVM threads creation. Shown as below, the script**<lab_root>/OOWHOL6427/DScripts/ex4.d** is to print the newly created thread when this java process created a new thread.

```
#!/usr/sbin/dtrace -qs

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}

proc:::lwp-create
/pid==$target/

{
        printf("%d/%d created %d\n",pid,tid,args[0]->pr_lwpid);
}

sysinfo::thread create:nthreads
/pid==$target/
{
        printf("pid is %d, and tid is %d\n",pid,tid);
}
```

Note: Both probes(proc:::lwp-create and sysinfo::thread_create:nthreads) will fire when the operating system is creating new threads, we can also use either of this two probes to monitor the java process.
args[0]->pr_lwpid in the lwp-create refer to the tid of newly created thread.

2. Open another terminal window, go to the directory "**<lab_root>/OOWHOL6427/DScripts**" and run the "ex4.d" script as following. (the number "1717" is the process id got from "jps" in the last step, your id maybe diferrent.)

```
labuser@Solaris11:~$  cd OOWHOL6427/DScripts/
labuser@Solaris11:~/OOWHOL6427/DScripts$ ./ex4.d -p 1717
Tracing... Hit Ctrl-C to end.
```

3. In the application terminal window, type "serial" again, and press **Enter** key, you will run the "`Exercise4---Run Serial Stream`" process, just as following:

```
labuser@Solaris11:~/OOWHOL6427$ gradle -q

Eager            :  Exercise2---Run EagerEvaluation
Lazy             :  Exercise2---Run LazyEvaluation
Pipeline         :  Exercise3---Run StreamPipeline
```

```
Serial           :  Exercise4---Run Serial Stream
Parallel         :  Exercise4---Run Parallel Stream
RecursiveCall    :  Exercise5---Run RecursiveCall
OpRecursiveCall  :  Exercise5---Run Optimized RecursiveCall
Print the your choice please:
>serial
working on one.....
working on two.....
working on three.....
working on four.....
working on five.....
working on six.....
working on seven.....
working on eight.....
```

4. In the DTrace script window, you will find nothing output.

5. In the application terminal window, type "parallel" again, and press **Enter** key, you will run the "`Exercise4---Run Parallel Stream`" process, just as following:

```
Eager            :  Exercise2---Run EagerEvaluation
Lazy             :  Exercise2---Run LazyEvaluation
Pipeline         :  Exercise3---Run StreamPipeline
Serial           :  Exercise4---Run Serial Stream
Parallel         :  Exercise4---Run Parallel Stream
RecursiveCall    :  Exercise5---Run RecursiveCall
OpRecursiveCall  :  Exercise5---Run Optimized RecursiveCall
Print the your choice please:
>parallel
working on five.....
working on three.....
working on six.....
working on four.....
working on seven.....
working on one.....
working on eight.....
working on two.....
```

6. In the DTrace script window, you will get some output like following:

```
pid is 1717, and tid is 13
1717/13 created 14
```

The output shows that in the Parallel application, new threads are created to perform the stream tasks.

7. We will use another DTrace script to trace Java method call stack when doing Stream Parallel process. Shown as below, the script**<lab_root>/OOWHOL6427/DScripts/ex4.1.d** is to trace methods call we interested in:

```
#!/usr/sbin/dtrace -qs


hotspot$target:::method-entry
/(stringof(copyin(arg1, 31)) == "Exercises/exercise4/Transaction")/
{
          jstack(20);
}
```

Note: The "jstack" function is used to trace JVM stack of Java method calls.

8. Open another terminal window, go to the directory "**<lab_root>/OOWHOL6427/DScripts**" and run the "ex4.1.d" script as following. (the number "1717" is the process id got from "jps" in the last step, your id maybe diferrent.)

```
labuser@Solaris11:~$  cd OOWHOL6427/DScripts/
labuser@Solaris11:~/OOWHOL6427/DScripts$ ./ex4.1.d -p 1717
```

9. In the application terminal window, type "parallel" again, and press **Enter** key, you will run the "`Exercise4---Run Parallel Stream`" process, just as following:

```
labuser@Solaris11:~/OOWHOL6427$ gradle -q

Eager            :  Exercise2---Run EagerEvaluation
Lazy             :  Exercise2---Run LazyEvaluation
Pipeline         :  Exercise3---Run StreamPipeline
Serial           :  Exercise4---Run Serial Stream
Parallel         :  Exercise4---Run Parallel Stream
RecursiveCall    :  Exercise5---Run RecursiveCall
OpRecursiveCall  :  Exercise5---Run Optimized RecursiveCall
Print the your choice please:
>parallel
working on five.....
working on three.....
working on six.....
working on four.....
working on seven.....
working on one.....
working on eight.....
working on two.....
```

10. In the DTrace script window, you will get some output like following:

```
             libjvm.so`__1cNSharedRuntimeTdtrace_method_entry6FpnKJavaThread_pnGMethod__i_+0xa4
              Exercises/exercise4/Transaction.process()V

Exercises/exercise4/StreamParallel.lambda$ParallelProcess$4(LExercises/exercise4/Transaction;)V
              Exercises/exercise4/StreamParallel$$Lambda$2.accept(Ljava/lang/Object;)V
              java/util/stream/ForEachOps$ForEachOp$OfRef.accept(Ljava/lang/Object;)V

java/util/ArrayList$ArrayListSpliterator.forEachRemaining(Ljava/util/function/Consumer;)V

java/util/stream/AbstractPipeline.copyInto(Ljava/util/stream/Sink;Ljava/util/Spliterator;)V
              java/util/stream/ForEachOps$ForEac
              0xffff80ffaa607c4d
              0xffff80ffaa607c4d
              …
```

The output may give hints that the JVM are using "Spliterator" to divide the work into parts and then assign these works to different working threads.

Step 3: Make you application more parallel

1. The parallelism factor of the Stream is controlled by the system property: "java.util.concurrent.ForkJoinPool.common.parallelism". In this step, we will modify the property to see the difference.

Open the file "build.gradle" in the directory of "**<lab_root>/OOWHOL6427/Applications**" with an editor you familiar, and modify the line marked with red as following, to remove the comment mark for this line. This property will tell the JVM to run Stream Parallelism to a factor of 8!

```
group 'com.oracle.oow2015'
version '1.0-SNAPSHOT'

apply plugin: 'java'
apply plugin: 'application'

sourceCompatibility = 1.8
mainClassName = "util.OOW2015Hol6427"

defaultTasks 'run'

repositories {
    mavenCentral()
}



run{
    standardInput = System.in
    jvmArgs '-XX:+ExtendedDTraceProbes'
    systemProperty 'java.util.concurrent.ForkJoinPool.common.parallelism', '8'
}

dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.11'
}
```

2. Rebuild the application. In the application terminal window, type "`stop`" to exit the application, and re-run it by type "`gradle -q`" again.

3. Open another terminal, and run the "jps" command as following:

```
labuser@solaris11:~# jps
1701 GradleMain
1722 OOW2015Hol6427
1742 Jps
labuser@solaris11:~#
```

By using the **jps**, we can find each Java processes with its process id. From the output above, We can tell that the process id of the above Java program on my machine is 1722(<pid>=1722, maybe different on your machine), for it has a name of *OOW2015Hol6427*. Then we can pick it out from other Java processes.

4. In this terminal window, go to the directory "**<lab_root>/OOWHOL6427/DScripts**" and run the "ex4.d" script as following. (the number "1722" is the process id got from "jps" in the last step, your id maybe diferrent.)

```
labuser@Solaris11:~$  cd OOWHOL6427/DScripts/
labuser@Solaris11:~/OOWHOL6427/DScripts$ ./ex4.d -p 1722
Tracing... Hit Ctrl-C to end.
```

5. In the application terminal window, type "parallel" again, and press **Enter** key, you will run the "`Exercise4---Run Parallel Stream`" process, just as following:

```
Eager            :  Exercise2---Run EagerEvaluation
Lazy             :  Exercise2---Run LazyEvaluation
Pipeline         :  Exercise3---Run StreamPipeline
Serial           :  Exercise4---Run Serial Stream
Parallel         :  Exercise4---Run Parallel Stream
RecursiveCall    :  Exercise5---Run RecursiveCall
OpRecursiveCall  :  Exercise5---Run Optimized RecursiveCall
Print the your choice please:
>parallel
working on five.....
working on three.....
working on six.....
working on four.....
working on seven.....
working on one.....
```

```
working on eight.....
working on two.....
```

6. In the DTrace script window, you will get some output like following (or similar):

```
pid is 1722, and tid is 22
1722/22 created 24
pid is 1722, and tid is 22
1722/22 created 25
pid is 1722, and tid is 22
1722/22 created 26
pid is 1722, and tid is 13
1722/13 created 22
pid is 1722, and tid is 13
1722/13 created 23
pid is 1722, and tid is 23
1722/23 created 27
pid is 1722, and tid is 27
1722/27 created 28
pid is 1722, and tid is 27
1722/27 created 29
```

Yes, by modifying the system property, you can use more threads to do your work!

We can get more interesting findings from the above output: the No.13 thread created two (No.22,23) threads; the No.22 thread created three (No.24,25,26) threads… and so on, which showed the split path of the parallel work.

Summary

In this exercise, you have created and run a DTrace script to trace the Java method calls and threads creation in your Java application. By observing the output of the DTrace script, we find that Stream will parallelize your application processing by creating as many threads as you configure and split the work into parts which are assigned to every thread.

Exercise 5: Observing Recursion Optimizing with DTrace (15 Minutes)

What is a recursive call

What is a recursive call? In computer programming it is the process of having a method continually call itself until a defined point of termination. The recursive call is particularly important for functional languages, as they often rely on recursive design and coding patterns.

What is a tail call

In computer science, a tail call is a subroutine call performed as the final action of a procedure. If a tail call might lead to the same subroutine being called again later in the call chain, the subroutine is said to be tail-recursive, which is a special case of recursion. Tail **recursion** (or **tail-end recursion**) is particularly useful, and often easy to handle in implementations.

What is Java support for tail call optimization

The truth is that Java platform **does not** implement any optimization for the recursive java methods, even for the tail recursive call (while many other languages support tail call optimization). In Java, because each recursive call uses some space on the Java stack. If your recursion is too deep, then it will result in StackOverflow Exception, depending upon the maximum allowed depth in the stack.

But with Java 8 new features, a few experts construct some patterns to do deep recursive calls without causing the StackOverflow Exceptions, by using the lazy evaluation properties of the Lambda expressions. In this exercise, we will write some DTrace scripts to trace the application to find out what it happen under the hood.

Background Information

Check the following Java Class named "SumRecursiveCall". The codes are very straight forward. The SumRecursive class is to implement the *Sum* function, which is defined for positive integers N by the equation SUM(N) = N + (N-1) + (N-2) + ... + 2 + 1. In the class We have two versions of recursive implementation. The method named "getSum1()" is implemented with formal Java method call. The method named "getSum2()" is implemented with Lambda expression.

The java source files (SumRecursiveCall.java and others) are located in **<lab_root>/OOWHOL10194/ Applications/src/main//java/Exercises/exercise5/**

```
public class SumRecursiveCall {

    private static int getSum1(int sum, int num) {
        if (num == 0) return sum;
        return getSum1(sum + num, num - 1);
    }

    private static RecursiveCall getSum2(final int sum,final int num) {
        if(num ==0)
```

```
            return lastCall(sum);
        else
            return getNextCall(() -> getSum2(sum + num, num - 1));
    }



    public static int SumOfInt1(int num) {
        int sum =  getSum1(0, num);
        System.out.println("The sum of int: "+sum);
        return sum;
    }

    public static int SumOfInt2(int num) {
        int sum =  getSum2(0, num).invoke();
        System.out.println("The sum of int: "+sum);
        return sum;
    }


}
```

Step 1: Run the application of Exercise 5

(if the application terminal window is still open after last exercise, you can reuse it, just skip to item 3.)
1. If the application is closed after the last exercise or you skip last exercises, please click the right mouse button on the desktop and choose *Open Terminal* to bring up a terminal window.

2. Go to the directory of **<lab_root>/OOWHOL6427/Applications/**, execute the command of **gradle -q** as following:

```
labuser@Solaris11:~/OOWHOL6427$  gradle -q

Eager            :  Exercise2---Run EagerEvaluation
Lazy             :  Exercise2---Run LazyEvaluation
Pipeline         :  Exercise3---Run StreamPipeline
Serial           :  Exercise4---Run Serial Stream
Parallel         :  Exercise4---Run Parallel Stream
RecursiveCall    :  Exercise5---Run RecursiveCall
OpRecursiveCall  :  Exercise5---Run Optimized RecursiveCall

Print the your choice please:
>
```

This **gradle** command runs a console program, which is interactive through the terminal. You can type the application name to choose the exercise you want to run. You can print the help information by press **Enter** key at any time, and stop it by enter the **"stop"** keyword.

3. Run the Exercise5 application with the input "`RecursiveCall 1000`". The number 1000 is the integer parameter N you pass in. You can try a larger number (30000) to get "StackOverflow" Exception.

```
Eager            :  Exercise2---Run EagerEvaluation
Lazy             :  Exercise2---Run LazyEvaluation
Pipeline         :  Exercise3---Run StreamPipeline
Serial           :  Exercise4---Run Serial Stream
Parallel         :  Exercise4---Run Parallel Stream
RecursiveCall    :  Exercise5---Run RecursiveCall
OpRecursiveCall  :  Exercise5---Run Optimized RecursiveCall

Print the your choice please:

>RecursiveCall 1000
The sum of int: 500500
>RecursiveCall 30000
Stack OverFlow Error!
```

4. Now try to use the optimized RecursiveCall version of this program with the input "`OpRecursiveCall 1000`". The number 1000 is the integer parameter N you pass in. You can try a larger number (30000) to find the result.

```
Eager            :  Exercise2---Run EagerEvaluation
Lazy             :  Exercise2---Run LazyEvaluation
Pipeline         :  Exercise3---Run StreamPipeline
Serial           :  Exercise4---Run Serial Stream
Parallel         :  Exercise4---Run Parallel Stream
RecursiveCall    :  Exercise5---Run RecursiveCall
OpRecursiveCall  :  Exercise5---Run Optimized RecursiveCall

Print the your choice please:
…..
>OpRecursiveCall 1000
The sum of int: 500500
>OpRecursiveCall 30000
The sum of int: 450015000
```

Step 2: Monitoring the program with JDK tools

1. Open another terminal, and run the "jps" command as following:

```
labuser@solaris11:~# jps
1701 GradleMain
1717 OOW2015Hol6427
1742 Jps
labuser@solaris11:~#
```

By using the **jps**, we can find each Java processes with its process id. From the output above, We can tell that the process id of the above Java program on my machine is 1717(<pid>=1717, maybe different on your machine), for it has a name of *OOW2015Hol6427*. Then we can pick it out from other Java processes.

Step 3: Writing a DTrace script to trace the call stacks of recursive methods

1. We will use "hotspot provider" to trace Java method calls to identify how Lambda expressions optimize the recursive Java calls. In this exercise, we will first use the following DTrace script to trace Java method calls. Shown as below, the script**<lab_root>/OOWHOL6427/DScripts/ex5.d** is to trace methods call we are interested in:

```
#!/usr/sbin/dtrace -qs

hotspot$target:::method-entry
/(stringof(copyin(arg1, 9)) == "Exercises")/
{
          self->indent++;
          printf("%*s %s %s.%s\n", self->indent, "", "->",
          stringof(copyin(arg1, arg2)), stringof(copyin(arg3,arg4)));
}

hotspot$target:::method-return
/(stringof(copyin(arg1, 9)) == "Exercises")/
{
          printf("%*s %s %s.%s\n", self->indent, "", "<-",
          stringof(copyin(arg1, arg2)), stringof(copyin(arg3,arg4)));
          self->indent--;
}
```

Note: this script is to trace method call entry and return point, print the class and method name in an indented format. The indented print format is controlled by variable "indent", Here, "self" is used (refer to exerciese1 on self), stating "indent" is a thread local variable to protect it from unwanted access.

2. Open another terminal window, go to the directory "**<lab_root>/OOWHOL6427/DScripts**" and run the "ex3.d" script as following. (the number "1717" is the process id got from "jps" in the last step, your id maybe diferrent.)

```
labuser@Solaris11:~$  cd OOWHOL6427/DScripts/
labuser@Solaris11:~/OOWHOL6427/DScripts$ ./ex5.d -p 1717
```

3. In the application terminal window, type "`RecursiveCall 5`" again, and press **Enter** key, you will run the "`RecursiveCall`" process, just as following:

```
Eager            :  Exercise2---Run EagerEvaluation
Lazy             :  Exercise2---Run LazyEvaluation
Pipeline         :  Exercise3---Run StreamPipeline
Serial           :  Exercise4---Run Serial Stream
Parallel         :  Exercise4---Run Parallel Stream
RecursiveCall    :  Exercise5---Run RecursiveCall
OpRecursiveCall  :  Exercise5---Run Optimized RecursiveCall

Print the your choice please:
>RecursiveCall 5
The sum of int: 15
```

4. In the DTrace script window, you will find some output, just like following:

```
  -> Exercises/exercise5/SumRecursiveCall.SumOfInt1
   -> Exercises/exercise5/SumRecursiveCall.getSum1
    -> Exercises/exercise5/SumRecursiveCall.getSum1
     -> Exercises/exercise5/SumRecursiveCall.getSum1
      -> Exercises/exercise5/SumRecursiveCall.getSum1
       -> Exercises/exercise5/SumRecursiveCall.getSum1
        -> Exercises/exercise5/SumRecursiveCall.getSum1
        <- Exercises/exercise5/SumRecursiveCall.getSum1
       <- Exercises/exercise5/SumRecursiveCall.getSum1
      <- Exercises/exercise5/SumRecursiveCall.getSum1
     <- Exercises/exercise5/SumRecursiveCall.getSum1
    <- Exercises/exercise5/SumRecursiveCall.getSum1
   <- Exercises/exercise5/SumRecursiveCall.getSum1
  <- Exercises/exercise5/SumRecursiveCall.SumOfInt1
```

5. In the application terminal window, type "`OpRecursiveCall 5`" again, and press **Enter** key, you will run the "`Optimized RecursiveCall`" process, just as following:

```
Eager            :  Exercise2---Run EagerEvaluation
Lazy             :  Exercise2---Run LazyEvaluation
Pipeline         :  Exercise3---Run StreamPipeline
Serial           :  Exercise4---Run Serial Stream
Parallel         :  Exercise4---Run Parallel Stream
RecursiveCall    :  Exercise5---Run RecursiveCall
OpRecursiveCall  :  Exercise5---Run Optimized RecursiveCall

Print the your choice please:
>RecursiveCall 5
The sum of int: 15
>OpRecursiveCall 5
The sum of int: 15
```

6. In the DTrace script window, you will find some output, just like following:

```
   -> Exercises/exercise5/SumRecursiveCall.SumOfInt2
   -> Exercises/exercise5/SumRecursiveCall.getSum2
    -> Exercises/exercise5/SumRecursiveCall$$Lambda$8.get$Lambda
```

```
    -> Exercises/exercise5/SumRecursiveCall$$Lambda$8.<init>
    <- Exercises/exercise5/SumRecursiveCall$$Lambda$8.<init>
   <- Exercises/exercise5/SumRecursiveCall$$Lambda$8.get$Lambda
   -> Exercises/exercise5/RecursiveCalls.getNextCall
   <- Exercises/exercise5/RecursiveCalls.getNextCall
  <- Exercises/exercise5/SumRecursiveCall.getSum2
  -> Exercises/exercise5/RecursiveCall.invoke
   -> Exercises/exercise5/RecursiveCall$$Lambda$10.test
    -> Exercises/exercise5/RecursiveCall.lambda$invoke$4
     -> Exercises/exercise5/RecursiveCall.result
     <- Exercises/exercise5/RecursiveCallall.resultnt1
    <- Exercises/exercise5/RecursiveCall.lambda$invoke$4
   <- Exercises/exercise5/RecursiveCall$$Lambda$10.test
   -> Exercises/exercise5/RecursiveCall$$Lambda$9.apply
    -> Exercises/exercise5/SumRecursiveCall$$Lambda$8.getNextCall
     -> Exercises/exercise5/SumRecursiveCall.lambda$getSum2$5
      -> Exercises/exercise5/SumRecursiveCall.getSum2
       -> Exercises/exercise5/SumRecursiveCall$$Lambda$8.get$Lambda
        -> Exercises/exercise5/SumRecursiveCall$$Lambda$8.<init>
        <- Exercises/exercise5/SumRecursiveCall$$Lambda$8.<init>
       <- Exercises/exercise5/SumRecursiveCall$$Lambda$8.get$Lambda
       -> Exercises/exercise5/RecursiveCalls.getNextCall
       <- Exercises/exercise5/RecursiveCalls.getNextCall
      <- Exercises/exercise5/SumRecursiveCall.getSum2
    <- Exercises/exercise5/SumRecursiveCall.lambda$getSum2$5
   <- Exercises/exercise5/SumRecursiveCall$$Lambda$8.getNextCall
  <- Exercises/exercise5/RecursiveCall$$Lambda$9.apply
  -> Exercises/exercise5/RecursiveCall$$Lambda$10.test
   -> Exercises/exercise5/RecursiveCall.lambda$invoke$4
    -> Exercises/exercise5/RecursiveCall.result
    <- Exercises/exercise5/RecursiveCall.result
   <- Exercises/exercise5/RecursiveCall.lambda$invoke$4
  <- Exercises/exercise5/RecursiveCall$$Lambda$10.test
  -> Exercises/exercise5/RecursiveCall$$Lambda$9.apply
   -> Exercises/exercise5/SumRecursiveCall$$Lambda$8.getNextCall
    -> Exercises/exercise5/SumRecursiveCall.lambda$getSum2$5
     -> Exercises/exercise5/SumRecursiveCall.getSum2
      -> Exercises/exercise5/SumRecursiveCall$$Lambda$8.get$Lambda
       -> Exercises/exercise5/SumRecursiveCall$$Lambda$8.<init>
       <- Exercises/exercise5/SumRecursiveCall$$Lambda$8.<init>
      <- Exercises/exercise5/SumRecursiveCall$$Lambda$8.get$Lambda
      -> Exercises/exercise5/RecursiveCalls.getNextCall
      <- Exercises/exercise5/RecursiveCalls.getNextCall
     <- Exercises/exercise5/SumRecursiveCall.getSum2
    <- Exercises/exercise5/SumRecursiveCall.lambda$getSum2$5
   <- Exercises/exercise5/SumRecursiveCall$$Lambda$8.getNextCall
  <- Exercises/exercise5/RecursiveCall$$Lambda$9.apply
  -> Exercises/exercise5/RecursiveCall$$Lambda$10.test
   -> Exercises/exercise5/RecursiveCall.lambda$invoke$4
    -> Exercises/exercise5/RecursiveCall.result
    <- Exercises/exercise5/RecursiveCall.result
   <- Exercises/exercise5/RecursiveCall.lambda$invoke$4
  <- Exercises/exercise5/RecursiveCall$$Lambda$10.test
  -> Exercises/exercise5/RecursiveCall$$Lambda$9.apply
   -> Exercises/exercise5/SumRecursiveCall$$Lambda$8.getNextCall
    -> Exercises/exercise5/SumRecursiveCall.lambda$getSum2$5
     -> Exercises/exercise5/SumRecursiveCall.getSum2
      -> Exercises/exercise5/RecursiveCalls.lastCall
       -> Exercises/exercise5/RecursiveCalls$1.<init>
       <- Exercises/exercise5/RecursiveCalls$1.<init>
      <- Exercises/exercise5/RecursiveCalls.lastCall
     <- Exercises/exercise5/SumRecursiveCall.getSum2
    <- Exercises/exercise5/SumRecursiveCall.lambda$getSum2$5
   <- Exercises/exercise5/SumRecursiveCall$$Lambda$8.getNextCall
  <- Exercises/exercise5/RecursiveCall$$Lambda$9.apply
  -> Exercises/exercise5/RecursiveCall$$Lambda$10.test
   -> Exercises/exercise5/RecursiveCall.lambda$invoke$4
    -> Exercises/exercise5/RecursiveCalls$1.result
```

```
     <- Exercises/exercise5/RecursiveCalls$1.result
    <- Exercises/exercise5/RecursiveCall.lambda$invoke$4
   <- Exercises/exercise5/RecursiveCall$$Lambda$10.test
   -> Exercises/exercise5/RecursiveCalls$1.result
   <- Exercises/exercise5/RecursiveCalls$1.result
  <- Exercises/exercise5/RecursiveCall.invoke
 <- Exercises/exercise5/SumRecursiveCall.SumOfInt2
```

From the output, you can easily see that the optimized version with Lambda expression just changes the recursive call into a loop, which avoid deep stack construction. If you are interested in the details of the implementation, you can browser the source code files under the "**<lab_root>/OOWHOL10194/Applications src/main//java/Exercises/exercise5**" directory. The main idea of it is to use the lazy evaluation property of Lambda expression to cache the recursive calls and change them to a loop.

Summary

In this exercise, you have created and run a DTrace script to trace the Java method calls and print the method call stacks. By observing the output of the DTrace script, we find that by using the Lambda expression lazy evaluation feature, we can change recursive method calls to a loop to reduce the StackOverFlow Exceptions.

Appendix A: Creating the Java code and the Gradle build files

Use the procedures in this appendix to create the directory structure, the .java files, and the build.gradle file used in the lab exercises.

Note: Directory names and file names are case-sensitive.

Create the Directory Structure

1. Decide which directory to use for your root directory. This directory will be referred to as **<lab_root>** in the exercises for this lab.

2. Under **<lab_root>**, create a nested subdirectory called "**OOWHOL10194/Applications/src/main/java"**.

3.Under "**<lab_root>/OOWHOL10194/Applications/src/main/java**", create two subdirectories, one named "Exercises" , the other named "util".

4. Under "**<lab_root>/OOWHOL10194/Applications/src/main/java/Exercises**", create four subdirectories, named "exercise2", "exercise3", "exercise4" and "exercise5" separately.

Create the Build Script file and Java Files

1. **Create the Gradle Build File**
Copy the code shown in Listing 1 and save it in a plain-text ASCII file named **<lab_root>/OOWHOL10194/Applications**/build.gradle:

```
group 'com.oracle.oow2015'
version '1.0-SNAPSHOT'

apply plugin: 'java'
apply plugin: 'application'

sourceCompatibility = 1.8
mainClassName = "util.OOW2015Hol6427"

defaultTasks 'run'

repositories {
    mavenCentral()
}

run{
    standardInput = System.in
    jvmArgs '-XX:+ExtendedDTraceProbes'
    //systemProperty 'java.util.concurrent.ForkJoinPool.common.parallelism', '8'
}

dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.11'
}
```

Listing 1. Code for the `build.gradle` file

2. **Create two Java files for "util"**

Copy the code shown in Listing 2 and save it in a plain-text ASCII file named
<lab_root>/OOWHOL10194/Applications/src/main/java/util/OOW2015Hol6427.java

```
package util;

import java.util.StringTokenizer;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

import Exercises.exercise2.LicenseEvaluate;
import Exercises.exercise3.StreamPipeline;
import Exercises.exercise5.SumRecursiveCall;
import Exercises.exercise4.StreamParallel;

public class OOW2015Hol6427 {

    private final static Executor pool = Executors.newFixedThreadPool(1);

    public static void main(String[] args) {

        OOW2015Hol6427 fws = new OOW2015Hol6427();
        Prompt shellPrompt = new Prompt();
        shellPrompt.setShell(fws);
        pool.execute(shellPrompt);
    }

    public void excuteEager() {
        LicenseEvaluate.eagerEvaluate(16, 2324444);
    }

    public void excuteLazy() {
        LicenseEvaluate.lazyEvaluate(16, 2343434);
    }

    public void streamPipeline() {
        StreamPipeline.getMyLuckyMovieNames();
    }

    public void noParallel(){
        StreamParallel.serialProcess();
    }

    public void arrayParallel(){
        StreamParallel.ParallelProcess();
    }

    public void recusiveCall(String input){
        try {
            SumRecursiveCall.SumOfInt1(parseInput(input));
        } catch (StackOverflowError e){
            System.out.println("Stack OverFlow Error!");
        }
    }
    public void opRecusiveCall(String input){
        SumRecursiveCall.SumOfInt2(parseInput(input));
    }


    private int parseInput(String input){
        StringTokenizer tokens = new StringTokenizer(input, " ");
        tokens.nextToken();
        String result = " ";
        if (tokens.hasMoreTokens()){
            result = tokens.nextToken().trim();
        }

        int reValue=0;
```

```
        try {
            reValue = Integer.parseInt(result);
        } catch( NumberFormatException e) {
            System.out.println("Cannot parse input to an Int value!");
            reValue =0;
        }
        return reValue;
    }
}
```

Listing 2. Code for the OOW2015Hol6427.java file

Copy the code shown in Listing 3 and save it in a plain-text ASCII file named **<lab_root>/OOWHOL10194/Applications/src/main/java/util/Prompt.java**

```
package util;

import java.io.*;

public class Prompt implements Runnable{

    private OOW2015Hol6427 OOW2015Hol6427;
    void setShell(OOW2015Hol6427 fws) {
        OOW2015Hol6427 = fws;
    }

    private void printPrompt() {
        System.out.print(">");
    }

    private void printOptions() {
        System.out.println();
        System.out.println("Eager            :  Exercise2---Run EagerEvaluation");
        System.out.println("Lazy             :  Exercise2---Run LazyEvaluation");
        System.out.println("Pipeline         :  Exercise3---Run StreamPipeline");
        System.out.println("Serial           :  Exercise4---Run Serial Stream");
        System.out.println("Parallel         :  Exercise4---Run Parallel Stream");
        System.out.println("RecursiveCall    :  Exercise5---Run RecursiveCall");
        System.out.println("OpRecursiveCall  :  Exercise5---Run Optimized RecursiveCall");

        System.out.println();
        System.out.println("Print the your choice please:");
    }

    public void run() {

        printOptions();
        printPrompt();
        boolean jump = false;
        while (!jump) {

            BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
            try {
                String input = in.readLine();
                if (input == null) {
                    jump = true;
                    input ="escape";
                }
                if (input.toLowerCase().startsWith("ea")) {
                    OOW2015Hol6427.excuteEager();
                }else if(input.toLowerCase().startsWith("la")) {
                    OOW2015Hol6427.excuteLazy();
                }else if(input.toLowerCase().startsWith("pi")) {
                    OOW2015Hol6427.streamPipeline();
                } else if(input.toLowerCase().startsWith("se")) {
                    OOW2015Hol6427.noParallel();
                } else if(input.toLowerCase().startsWith("pa")) {
                    OOW2015Hol6427.arrayParallel();
```

```
                } else if(input.toLowerCase().startsWith("re")) {
                    OOW2015Hol6427.recusiveCall(input.trim());
                }   else if(input.toLowerCase().startsWith("op")) {
                        OOW2015Hol6427.opRecusiveCall(input.trim());
                }else if(input.startsWith("stop") || input.startsWith("exit")) {
                    System.out.println("Bye!");
                    jump = true;
                }else {
                    printOptions();
                }

                printPrompt();

            } catch (Exception ioe) {
                ioe.printStackTrace();
            }
        }

        System.out.println("");
        System.exit(0);
    }
}
```

Listing 3. Code for the Prompt.java file

3. **Create one Java file for Exercise 2**

Copy the code shown in Listing 2 and save it in a plain-text ASCII file named **<lab_root>/OOWHOL10194/Applications/src/main/java/Exercises/exercise2/LicenseEvaluate.java**:

```
package Exercises.exercise2;
import java.util.function.Supplier;

public class LicenseEvaluate {

    public static boolean isOldEnough(int age){
        return age >= 18 ;
    }
    public static boolean isDriverLicenseValid(final int licensecode) {
        consumingProcess();
        return true;
    }

    private static boolean isDriverLegal1(final boolean input1,final boolean input2){
        return input1&&input2;
    }

    private static boolean isDriverLegal2(final Supplier<Boolean> input1,final Supplier<Boolean>
input2){
        return input1.get() && input2.get();
    }

    public static void eagerEvaluate(final int age, final int code){
        boolean result = isDriverLegal1(isOldEnough(age), isDriverLicenseValid(code));
        if (result) System.out.println("It's legal!");
        else System.out.println("It is illegal!");
    }

    public static void lazyEvaluate(final int age, final int code){
        boolean result = isDriverLegal2(() -> isOldEnough(age), () ->
isDriverLicenseValid(code));
        if (result) System.out.println("It's legal!");
        else System.out.println("It is illegal!");
    }

    private static void consumingProcess() {
```

```
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Listing 4. Code for the `LicenseEvaluate.java` file

4. **Create the two Java files for Exercise 3**

Copy the code shown in Listing 5 and save it in a plain-text ASCII file named **<lab_root>/OOWHOL10194/Applications/src/main/java/Exercises/exercise3/StreamPipeline.java**:

```
package Exercises.exercise3;

import java.util.List;

public class StreamPipeline {
    public static void getMyLuckyMovieNames() {
        List<Movie> popularMovies = Movie.getPopularMovies();
        String myLuckyMovie = popularMovies.stream()
                .filter(Movie::isRecent)
                .filter(Movie::isVeryPopular)
                .map(Movie::getUppperName)
                .findFirst()
                .get();

        System.out.println("I found a good movie: "+myLuckyMovie);
    }
}
```

Listing 5. Code for the `StreamPipeline.java` file

Copy the code shown in Listing 6 and save it in a plain-text ASCII file named **<lab_root>/OOWHOL10194/Applications/src/main/java/Exercises/exercise3/Movie.java**:

```
package Exercises.exercise3;

import java.util.ArrayList;
import java.util.List;

public class Movie {

    private String name;
    private int releaseYear;
    private double score;
    public Movie(String m_name,int m_year,double m_score) {
        name = m_name;
        releaseYear = m_year;
        score = m_score;
    }

    public static boolean isVeryPopular(Movie movie) {
        return movie.score>9.6;
    }

    public static boolean isRecent(Movie movie) {
        return movie.releaseYear>2008;
    }

    public static String getUppperName(Movie movie) {
        return movie.name.toUpperCase();
    }

    public static List<Movie> getPopularMovies() {
        List<Movie> popularMovies = new ArrayList<Movie>();
        popularMovies.add(new Movie("Boyhood",2014,10.0));
```

```
        popularMovies.add(new Movie("Pan's Labyrinth",2006,9.8));
        popularMovies.add(new Movie("Carol",2015,9.8));
        popularMovies.add(new Movie("Hoop Dreams",1994,9.8));
        popularMovies.add(new Movie("4 Months, 3 Weeks and 2 Days",2008,9.7));
        popularMovies.add(new Movie("12 Years A Slave",2013,9.7));
        popularMovies.add(new Movie("Dr. Strangelove",1964,9.7));
        popularMovies.add(new Movie("Ratatouille",2007,9.6));
        popularMovies.add(new Movie("Gravity",2013,9.6));
        popularMovies.add(new Movie("The Social Network",2010,9.5));
        popularMovies.add(new Movie("Zero Dark Thirty",2012,9.5));
        popularMovies.add(new Movie("Zero Dark Thirty",2011,9.5));
        return popularMovies;
    }

}
```

Listing 6. Code for the `Movie.java` file

5. **Create two Java files for Exercise 4**

Copy the code shown in Listing 7 and save it in a plain-text ASCII file named **<lab_root>/OOWHOL10194/Applications/src/main/java/Exercises/exercise4/Transaction.java**:

```
package Exercises.exercise4;

public class Transaction {
    private String name;

    public Transaction(String t_name){
        name = t_name;
    }
    public void process() {
        System.out.println("working on "+name+".....");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Listing 7. Code for the `Transaction.java` file

Copy the code shown in Listing 8 and save it in a plain-text ASCII file named **<lab_root>/OOWHOL10194/Applications/src/main/java/Exercises/exercise4/StreamParallel.java**:

```
package Exercises.exercise4;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.stream.Stream;

public class StreamParallel {
    static List<Transaction> transactions = new ArrayList<Transaction>();
    static {
        transactions.add(new Transaction("one"));
        transactions.add(new Transaction("two"));
        transactions.add(new Transaction("three"));
        transactions.add(new Transaction("four"));
        transactions.add(new Transaction("five"));
        transactions.add(new Transaction("six"));
        transactions.add(new Transaction("seven"));
        transactions.add(new Transaction("eight"));
```

```
    }

    public static void ParallelProcess(){
        transactions.stream().parallel().forEach(t -> t.process());
    }

    public static void serialProcess(){
        transactions.stream().forEach(t -> t.process());
    }
}
```

Listing 8. Code for the `StreamParallel.java` file

6. **Create three Java files for Exercise 5**

Copy the code shown in Listing 9 and save it in a plain-text ASCII file named **<lab_root>/OOWHOL10194/Applications/src/main/java/Exercises/exercise4/RecursiveCall.java**:

```
package Exercises.exercise5;

import java.util.stream.Stream;

@FunctionalInterface
public interface RecursiveCall {
    RecursiveCall getNextCall();

    default int result() {return 0;}
    default int invoke() {
        return Stream.iterate(this, RecursiveCall::getNextCall)
                .filter(u -> u.result() !=0)
                .findFirst()
                .get()
                .result();
    }
}
```

Listing 9. Code for the `RecursiveCall.java` file

Copy the code shown in Listing 10 and save it in a plain-text ASCII file named **<lab_root>/OOWHOL10194/Applications/src/main/java/Exercises/exercise4/SumRecursiveCall.java**:

```
package Exercises.exercise5;

import static Exercises.exercise5.RecursiveCalls.getNextCall;
import static Exercises.exercise5.RecursiveCalls.lastCall;

public class SumRecursiveCall {

    private static int getSum1(int sum, int num) {
        if (num == 0) return sum;
        return getSum1(sum + num, num - 1);
    }

    private static RecursiveCall getSum2(final int sum,final int num) {
        if(num ==0)
            return lastCall(sum);
        else
            return getNextCall(() -> getSum2(sum + num, num - 1));
    }



    public static int SumOfInt1(int num) {
        int sum =  getSum1(0, num);
        System.out.println("The sum of int: "+sum);
        return sum;
```

```
    }

    public static int SumOfInt2(int num) {
        int sum =  getSum2(0, num).invoke();
        System.out.println("The sum of int: "+sum);
        return sum;
    }
}
```

Listing 10. Code for the `SumRecursiveCall.java` file

Copy the code shown in Listing 11 and save it in a plain-text ASCII file named **<lab_root>/OOWHOL10194/Applications/src/main/java/Exercises/exercise4/RecursiveCalls.java**:

```
package Exercises.exercise5;

public class RecursiveCalls {
    public static  RecursiveCall getNextCall(final RecursiveCall nextCall){
        return nextCall;
    }

    public static RecursiveCall lastCall(final int value) {
        return new RecursiveCall() {
            @Override public int result() {return value;}
            @Override public RecursiveCall getNextCall() {return null;}
        };
    }
}
```

Listing 11. Code for the `RecursiveCalls.java` file

Summary

You have successfully completed the " Uncover JDK 8 Secrets Using DTrace on Oracle Solaris 11" hands-on lab! You have learned how to use the Oracle Solaris 11 DTrace scripts to find out how a Java Virtual Machine (JVM) 8 implements some new features, including Streams Pipeline, Stream Parallelism, Lambda Lazy Evaluation, and Recursion Optimizing.

See Also

- Oracle Solaris 11.2 Dynamic Tracing Guide
- Observing and Optimizing your Application with DTrace
- DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD
- Java 8 Lambda Expression Tutorial
- “Java 8 Lambdas” by Richard Warburton
- “Functional.Programming.in.Java” by Venkat Subramaniam

About the Authors

Yu Wang presently works for Oracle’s ISV Engineering group as a Principal Software Engineer. His duties include supporting local ISVs and evangelizing about Oracle Solaris and Java technologies.

Xiaosong (Chris) Zhu is a Principal Software Engineer working for Oracle’s ISV Engineering group. She is concentrated on Solaris and C/C++. Her duties include doing Solaris evangelizing and supporting local ISVs to run C/C++ applications best on Oracle Solaris and SPARC servers.

Wen-sheng Liu is a Principal Software Engineer working for Oracle’s ISV Engineering group. He is concentrated on Solaris and C/C++. Her duties include doing Solaris evangelizing and supporting local ISVs to run C/C++ applications best on Oracle Solaris and SPARC servers.